البرمجة بلغة بايثون



تأليف
ليزا تاغليفيري

البرمجة بلغة بايثون

تعلم البرمجة وكتابة البرامج وتنقيحها بلغة بايثون

تأليف

ليزا تاغليفيري

ترجمة

محمد بغات

عبد اللطيف ايمش

تحرير

جميل بيلوني

تصميم الغلاف

فرج الشامي

عن الناشر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.



أكاديمية حسوب

تهدف أكاديمية حسوب إلى توفير مقالات ودروس عالية الجودة حول مجالات مُختلفة وبلغة عربية فصيحة.

تقدم أكاديمية حسوب دورات شاملة بجودة عالية عن تعلم البرمجة بأحدث تقنياتها تعتمد على التطبيق العملي، مما يؤهل الطالب لدخول سوق العمل بثقة.

تتكامل الأكاديمية مع موسوعة حسوب، التي توفر توثيقًا عربيًا شاملًا مدعمًا بالأمثلة للغات البرمجة.

باب المُساهمة في الأكاديمية مفتوح لكل من يرى في نفسه القدرة على توفير مقالات أو كتب أو مسارات عالية الجودة.

Academy.hsoub.com



شركة حسوب

تهدف حسوب لتطوير الويب العربي وخدمات الإنترنت عن طريق توفير حلول عملية وسهلة الاستخدام لتحديات مختلفة تواجه المستخدمين في العالم العربي.

تشجع حسوب الشباب العربي للدخول إلى سوق العمل عن بعد بتوفيرها منصات عربية للعمل عن بعد، مستقل وخمسات؛ إضافةً إلى موقع بعيد، وكما أنها توفر خدمات للنقاشات الهادفة في حسوب I/O وخدمة رفع الصور عبر موقع صور.

يعمل في حسوب فريق شاب وشغوف من مختلف الدول العربية. ويمكن معرفة المزيد عن شركة حسوب والخدمات التي تقدمها بزيارة موقعها.

Hsoub.com

جدول المحتويات

تقديم..15
1. كيفية استخدام هذا الكتاب..............................17
2. ماذا بعد هذا الكتاب...18

مدخل تعريفي إلى لغة بايثون...............20
1. تاريخ بايثون..22
2. مميزات لغة بايثون...22
3. أين تُستخدَم بايثون؟.......................................23
4. لماذا بايثون وليس غيرها؟.................................24
5. خلاصة الفصل..27

تثبيت بايثون وإعداد بيئة العمل.............28
1. ويندوز...30
2. أوبنتو...41
3. دبيان..47
4. CentOS...54
5. macOS...60

سطر أوامر بايثون التفاعلي...................70
1. فتح سطر الأوامر التفاعلي.................................71

2. العمل في سطر أوامر بايثون التفاعلي.................... 73
3. تعدُّد الأسطر.. 74
4. استيراد الوحدات... 75
5. الخروج من سطر أوامر بايثون التفاعلي................. 77
6. الاطلاع على التاريخ.. 78
7. خلاصة الفصل... 79

التعليقات واستخداماتها...................... 80

1. صياغة التعليقات... 81
2. التعليقات الكتلية... 83
3. التعليقات السطرية.. 84
4. تعليق جزء من الشيفرة بدواعي الاختبار والتنقيح..... 85
5. خلاصة الفصل... 87

المتغيرات واستخداماتها...................... 88

1. فهم المتغيرات... 89
2. قواعد تسمية المتغيرات.................................... 93
3. تغيير قيم المتغيرات.. 95
4. الإسناد المتعدد (Multiple Assignment)................. 96
5. المتغيرات العامة والمحلية................................ 97
6. خلاصة الفصل... 102

أنواع البيانات والتحويل بينها................ 103

1. خلفية عامة.. 104
2. الأعداد... 105
3. القيم المنطقية.. 107

4. السلاسل النصية..109
5. القوائم (Lists)...110
6. الصفوف (Tuples)..111
7. القواميس (Dictionaries)...............................112
8. التحويل بين أنواع البيانات...............................113
9. خلاصة الفصل...125

السلاسل النصية والتعامل معها..........127

1. إنشاء وطباعة السلاسل النصية.........................128
2. آلية فهرسة السلاسل النصية.............................129
3. تقسيم السلاسل النصية...................................131
4. جمع السلاسل النصية.....................................135
5. تكرار السلاسل النصية....................................136
6. تخزين السلاسل النصية في متغيرات...................136
7. دوال السلاسل النصية....................................137
8. دوال الإحصاء...143
9. خلاصة الفصل...146

مدخل إلى تنسيق النصوص.................147

1. الصياغة المختزلة..148
2. علامات الاقتباس..148
3. كتابة النص على أكثر من سطر.........................149
4. تهريب المحارف..150
5. السلاسل النصية الخام....................................152
6. استخدام المُنسِّقات...153
7. تحديد نوع القيمة...161

8. إضافة حواشي...163
9. استخدام المتغيرات..165
10. خلاصة الفصل..166

العمليات الحسابية............................167

1. العاملات...168
2. الجمع والطرح...170
3. العمليات الحسابية الأحادية...............................171
4. الضرب والقسمة..172
5. عامل باقي القسمة (Modulo)..............................174
6. القوة (Power)...174
7. أسبقية العمليات الحسابية.................................175
8. عامل الإسناد (Assignment Operators)................176
9. إجراء العمليات الرياضية عبر الدوال.....................178
10. خلاصة الفصل..186

العمليات المنطقية (البوليانية)............187

1. عامل الموازنة...188
2. العاملات المنطقية..191
3. جداول الحقيقة (Truth Tables)...........................194
4. استعمال المنطق للتحكم في مسار البرنامج...........196
5. خلاصة الفصل..197

النوع List: مدخل إلى القوائم...............198

1. فهرسة القوائم (Indexing Lists).........................200
2. تعديل عناصر القائمة.....................................202

3. تقطيع القوائم (Slicing Lists)............................202
4. تعديل القوائم بالعوامل....................................205
5. إزالة عنصر من قائمة.......................................207
6. بناء قوائم من قوائم أخرى موجودة......................208
7. استخدام توابع القوائم.....................................209
8. فهم كيفية استعمال List Comprehensions...........217
9. خلاصة الفصل...223

النوع Tuple: فهم الصفوف..................225

1. فهرسة الصفوف..227
2. تقطيع قيم صف...229
3. إضافة بنى صف إلى بعضها..................................231
4. دوال التعامل مع الصفوف...................................233
5. كيف تختلف بنى الصفوف عن القوائم....................235
6. خلاصة الفصل...236

النوع Dictionary: فهم القواميس..........237

1. الوصول إلى عناصر قاموس..................................239
2. تعديل القواميس...243
3. حذف عناصر من القاموس.....................................247
4. خلاصة الفصل...249

التعليمات الشرطية.........................250

1. التعليمة if..251
2. التعليمة else...253
3. التعليمة else if..254

4. تعليمات if المتشعبة..257
5. خلاصة الفصل..262

المهام التكرارية: مدخل إلى الحلقات.....263

1. حلقة التكرار while..264
2. حلقة التكرار for..272
3. التحكم بحلقات التكرار......................................282
4. خلاصة الفصل..287

الدوال: تعريفها واستعمالها...............288

1. تعريف دالة..289
2. المعاملات: تمرير بيانات للدوال..........................291
3. الوسائط المسمَّاة..293
4. القيم الافتراضية للوسائط.................................295
5. إعادة قيمة..296
6. استخدام ()main دالةً رئيسيةً............................299
7. استخدام args* و kwargs**..............................305
8. ترتيب الوسائط...310
9. استخدام args* و kwargs**..............................311
10. خلاصة الفصل..313

الوحدات: استيرادها وإنشاؤها.............314

1. تثبيت الوحدات...316
2. استيراد الوحدات...317
3. استيراد عناصر محدَّدة....................................320
4. الأسماء المستعارة في الوحدات..........................321

5. كتابة وحدات مخصَّصة واستيرادها......................322
6. الوصول إلى الوحدات من مجلد آخر....................326
7. خلاصة الفصل...329

بناء الأصناف واستنساخ الكائنات..........330

1. الأصناف...331
2. الكائنات..332
3. الباني (Constructor)......................................334
4. العمل مع عدة كائنات.......................................337
5. فهم متغيرات الأصناف والنسخ..........................338
6. العمل مع متغيرات الصنف والنسخة معًا................343
7. خلاصة الفصل...344

مفهوم الوراثة في البرمجة.................346

1. ما هي الوراثة؟...347
2. الأصناف الأساسية...348
3. الأصناف الفرعية...350
4. إعادة تعريف توابع الصنف الأساسي....................353
5. الدالة ()super وفائدتها في الوراثة.......................355
6. الوراثة المُتعدِّدة (Multiple Inheritance)..............358
7. خلاصة الفصل...360

التعددية الشكلية وتطبيقاتها............361

1. ما هي التعددية الشكلية (Polymorphism)؟..........362
2. إنشاء أصناف متعددة الأشكال..........................363
3. التعددية الشكلية في توابع الأصناف....................365

4. التعددية الشكلية في الدوال............................366
5. خلاصة الفصل...368

تنقيح الشيفرات: استخدام منقِّح بايثون..369

1. تشغيل منقح بايثون تفاعليًا...............................370
2. استخدام المنقح للتنقل ضمن البرنامج.................372
3. نقاط التوقف...376
4. دمج pdb مع البرامج......................................379
5. تعديل تسلسل تنفيذ البرنامج............................380
6. جدول بأوامر pdb الشائعة...............................384
7. الوحدة code: تنقيح الشيفرات من سطر الأوامر 385
8. الوحدة Logging: التنقيح بالتسجيل وتتبع الأحداث. 390
9. خلاصة الفصل...403

إصدارات بايثون: الإصدار 3 مقابل 2........404

1. بايثون 2..405
2. بايثون 3..405
3. بايثون 2.7..406
4. الاختلافات الأساسية بين الإصدارات...................407
5. نقاط أخرى يجب أخذها بالحسبان......................410
6. ترحيل شيفرة بايثون 2 إلى بايثون 3....................411
7. تعرف على الاختلافات بين بايثون 2 و بايثون 3......413
8. تحديث الشيفرة...417
9. التكامل المستمر (Continuous Integration)..........418
10. خلاصة الفصل..419

ت

تقديم

سطع نجم لغة البرمجة بايثون في الآونة الأخيرة حتى بدأت تزاحم أقوى لغات البرمجة في الصدارة وذاك لمزايا هذه اللغة التي لا تنحصر أولها سهولة كتابة وقراءة شيفراتها حتى أضحت الخيار الأول بين يدي المؤسسات الأكاديمية والتدريبية لتدريسها للطلاب الجدد الراغبين في الدخول إلى مجال علوم الحاسوب والبرمجة. أضف إلى ذلك أن بايثون لغةً متعدِّدة الأغراض والاستخدامات، لذا فهي دومًا الخيار الأول في شتى مجالات علوم الحاسوب الصاعدة مثل الذكاء الصنعي وتعلم الآلة وعلوم البيانات وغيرها، كما أنَّها مطلوبة بشدة في سوق العمل وتعتمدها كبرى الشركات التقنية.

جاء هذا الكتاب المترجم عن كتاب «How to code in Python» لصاحبته ليزا تاغليفيري (Lisa Tagliaferri) ليشرح المفاهيم البرمجية الأساسية بلغة بايثون، ونأمل أن يكون إضافةً نافعةً للمكتبة العربيَّة وأن يفيد القارئ العربي في أن يكون منطلقًا للدخول إلى عالم البرمجة من أوسع أبوابه.

نظرًا لكونه متوفرًا على هيئة كتاب إلكتروني، فبإمكانك استخدام كتاب «البرمجة بلغة بايثون» مرجعًا تعليميًا مفتوحًا، وبالتالي يمكن استخدامه في أي فصل دراسي سواءً في المدرسة أو الجامعة، كما يمكن توفير هذا الكتاب الإلكتروني للعامة في المكتبات.

يمكن استخدام هذا الكتاب الإلكتروني بعدة طرائق، وسوف نوضح في هذا التقديم كيفيَّة التعامل مع الكتاب، وكيف يمكن للمعلمين والطلاب استخدام الكتاب في فصولهم الدراسية، وكيف يمكن لأمناء المكتبات العامة والجامعية توفير هذا الكتاب الإلكتروني ليكون مرجعًا تعليميًّا. أخيرًا، بالنسبة للقارئ الذي أنهى الكتاب ويريد توجيهًا حول ما يجب أن يتعلمه لاحقًا، فقد أضفنا بعض المراجع الإضافية في آخر هذا القسم.

1. كيفية استخدام هذا الكتاب

صمِّم هذا الكتاب بطريقة سلسة ومنطقيَّة. ورغم أنَّه مُرتب ترتيبًا يناسب المطور المبتدئ، إلا أنَّه ليس عليك التقيد بالترتيب: ابدأ حيث شئت، وأقرأه بالترتيب الذي يناسب احتياجاتك. بعد إنهاء الكتاب، يمكنك استخدامه مرجعًا.

إذا قرأت الكتاب بالترتيب، فستبدأ رحلتك في بايثون من مقدمة عامة حول اللغة لمن لا يعرفها، بعد ذلك، ستتعلم إعداد بيئة برمجة على الجهاز المحلي أو على الخادم، وستبدأ تعلم البنية العامة لشيفرة بايثون، وصياغتها، وأنواع البيانات فيها. في ثنِيّات الطريق، ستتعلم أساسيات المنطق الحسابي في بايثون، وهي مهارات مفيدة حتى في لغات البرمجة الأخرى. سنركز في بداية الكتاب على كتابة السكربتات في بايثون، ثم سنقدِّم بالتدريج مفاهيم البرمجة الكائنية لمساعدتك على كتابة شيفرات أكثر مرونة وتعقيدًا، مع تجنب التكرار. وفي نهاية الكتاب، ستتعلم كيفية تنقيح شيفرة بايثون، وسنختم بفصل عن الاختلافات الرئيسية بين بايثون 3 والإصدارات السابقة وكيفية ترحيل شيفرة بايثون من الإصدار بايثون 2 إلى بايثون 3.

ا. استخدام الكتاب في الفصل الدراسي

إذا كنت طالبًا، فيمكنك إطلاع معلمك أو أستاذك أو قسم الحوسبة على هذا الكتاب

الإلكتروني المجاني. قد يوجد في مدرستك أو جامعتك مستودعًا أو مكتبةً مفتوحةً للمراجع التعليمية والتي يمكنك فيها إتاحة هذا الكتاب للطلاب أو المعلمين. يمكنك أيضًا مشاركة هذا الكتاب الإلكتروني مع الأندية والمجموعات التي تنتمي إليها والتي قد تكون مهتمة بتعلم البرمجة بلغة بايثون. وإضافة إلى الأندية والبرامج الخاصة بعلوم الحاسوب، يمكن أن يستفيد أيضًا من هذا الكتاب الأشخاص الذين يدرسون علم البيانات والإحصاء والعلوم الإنسانية الرقمية.

إذا كنت معلمًا تُدرِّس أو تشرف على ورشات برمجية تعلِّم فيها بايثون، فيمكنك استخدام هذا الكتاب التعليمي المفتوح مجانًا مع طلابك. يمكنك اتباع ترتيب فصول الكتاب، أو يمكنك انتقاء الفصول التي تناسب المُقرَّر الذي تُدرِّسه وفق الترتيب المناسب لك. يمكنك أيضًا تكميل هذا الكتاب الرقمي بالكثير من الدروس والمقالات من أكاديمية حسوب أو غيرها، ويمكنك أيضًا استخدام موسوعة حسوب مرجعًا للغة بايثون (وغيرها من اللغات).

ب. إضافة الكتاب إلى مكتبتك

إذا كنت أمين مكتبة، فيمكنك إضافة كتاب «البرمجة بلغة بايثون» إلى فهرس مكتبتك. صحيح أنَّ ليس كل الناس مهتمين بالبرمجة، إلا أنَّ تعلم بعض مبادئ البرمجة يمكن أن يكون مفيدًا في الحياة المهنية، ويساعد على تخفيض الأميَّة الرقميَّة.

2. ماذا بعد هذا الكتاب

عند الانتهاء من هذا الكتاب، يمكنك الاطلاع على العديد من المقالات العملية في أكاديمية حسوب. أثناء ذلك، يمكنك التنقل بين توثيق بايثون في موسوعة حسوب وفصول هذا الكتاب.

يمكن لأيِّ شخص ملم بالبرمجة أن يساهم في المشاريع مفتوحة المصدر. البرامج مفتوحة المصدر هي برامج متاحة للاستخدام وإعادة التوزيع والتعديل دون قيود. تساعد المساهمة في

المشـاريع مفتوحـة المصـدر على تحسـين الـبرامج، عـبر ضـمان تمثيلهـا لقاعـدة عريضـة من المسـتخدمين. عنـدما يسـاهم المسـتخدمون في المشـاريع مفتوحـة المصـدر، سـواء عـبر كتابـة الشـيفرة، أو التوثيـق، أو صـيانة المجلـدات، فـإنهم يـوفرون قيمـة مضـافة للمشـروع، ومجتمـع المطورين على العموم.

للحصـول على مراجـع إضـافية عن بـايثون، أو للمشـاركة في نقاشـات مـع الآخـرين، يمكنـك الاطلاع على المقالات والأسئلة والدروس عن بايثون في الأكاديمية.

إذا كنت مهتمًا بتعلم تطوير تطبيقات الويب أو تطبيقات الجوال، أو تعلم لغات محدَّدة مثل روبي وجافاسـكربت، فـاطلع على قسـم الـدورات في الأكاديميـة، كمـا يمكنـك تصـفح موسـوعة حسوب لأجل قراءة توثيقات عدد كبير من لغات البرمجة باللغة العربية.

جميل بيلوني

01 - يوليو - 2020

1

مدخل تعريفي إلى لغة بايثون

بايثون لغةٌ سهلة القراءة للغاية ومتنوعة ومتعددة الاستخدامات، واسمها مستوحى من مجموعة كوميدية بريطانية باسم «Monty Python»، وكان أحد الأهداف الأساسية لفريق تطوير بايثون هو جعل اللغة مرحةً وسهلة الاستخدام، وإعدادها بسيطٌ، وطريقة كتابتها مباشرة وتعطيك تقريرًا مباشرًا عند حدوث أخطاء، وهي خيارٌ ممتازٌ للمبتدئين والوافدين الجدد على البرمجة. لغة بايثون هي لغة متعدِّدة الاستعمالات، وتدعم مختلف أنماط البرمجة مثل كتابة السكربتات والبرمجة كائنيَّة التوجه (object-oriented)، وهي مناسبةٌ للأغراض العامة، واستعمالها يتزايد في سوق العمل إذ تعتمدها منظماتٌ مثل «United Space Alliance» (شركة في مجال إرسال مركبات فضائية وتتعاقد معها ناسا) و «Industrial Light & Magic» (أستوديو للتأثيرات السينمائية وللرسوم المتحركة)، وتوفِّر بايثون قدراتٍ كثيرةٍ لمن يريد تعلم لغة برمجة جديدة.

طوِّرَت اللغة في نهاية الثمانينات من القرن الماضي، ونُشِرَت أوّل مرة في عام 1991، طُوِّرَت بايثون من قِبل Guido van Rossum، وهو عضوٌ نشطٌ للغاية في المجتمع. وتعدُّ بايثون على أنَّها بديلٌ عن لغة ABC، وأوّل إصدار منها كان يتضمن التعامل مع الاستثناءات (exception handling) والدوال والأصناف (classes) مع إمكانية الوراثة فيها. وعدما أُنشِئ منتدى محادثة في Usenet باسم comp.lang.python في 1994، فبدأت قاعدة مستخدمي بايثون بالنمو، مما مهّد الطريق لها لتصبح واحدة من أكثر لغات البرمجة شيوعًا وخصوصًا لتطوير البرمجيات مفتوحة المصدر.

1. تاريخ بايثون

ظهرت لغة بايثون في أواخر الثمانينيات على يد غيدو فان روسوم (Guido van Rossum)، وقد عُدَّت خليفةً للغة ABC. كما استفادت بايثون من الكثير من اللغات السابقة لها، مثل Modula-3 و C و C++ و Algol-68 و SmallTalk، وغيرها من اللغات.

نُشِر الإصدار 2.0 من لغة بايثون عام 2000، وقد قدَّم العديد من الميزات الجديدة، مثل القوائم الفهميَّة (List Comprehensions)، ونظام كنس المُهملات (garbage collection). وظهر في عام 2008 الإصدار بايثون 3.0، والذي شكَّل طفرةً في اللغة، بيْد أنَّه لم يكن متوافقًا تمامًا مع الإصدارات السابقة، لذلك قرر فريق التطوير الاستمرار في دعم إصدار أخير من سلسلة بايثون 2.x، وهو بايثون 2.7 حتى عام 2020.

2. مميزات لغة بايثون

تتميز بايثون بعدة أمور عن غيرها من لغات البرمجة، منها:

- **سهولة التعلُّم**: يسهل تعلم لغة بايثون، إذ تتألف من عدد قليل من الكلمات المفتاحيَّة، وتتميز بصياغة بسيطة وواضحة.
- **المقروئية**: شيفرة لغة بايثون واضحة ومنظمة وسهلة القراءة.
- **سهلة الصيانة**: شيفرة بايثون سهلة الصيانة إلى حد بعيد.
- **مكتبة قياسية واسعة**: تحتوي مكتبة بايثون القياسية على عدد كبير من الحزم المحمولة التي تتوافق مع أنظمة يونكس وويندوز وماك.
- **الوضع التفاعلي**: تدعم بايثون الوضع التفاعلي، مما يتيح إمكانيَّة تنفيذ الشيفرات مباشرةً على سطر الأوامر وتنقيحها.

- **متعدِّدة المنصات:** يمكن تشغيل لغة بايثون على طيف واسع من المنصات والأجهزة، مع الاحتفاظ بنفس الواجهة على جميع تلك المنصات.
- **التوسُّعية:** من أهم مميزات بايثون، هو توفرها على عدد هائل من الوحدات، التي يمكنها توسيع قدرات اللغة في كل مجالات التطوير، مثل تحليل البيانات والرسوميات ثنائية وثلاثية الأبعاد، وتطوير الألعاب، والأنظمة المدمجة، والبحث العلمي، وتطوير المواقع وغيرها من المجالات.
- **قواعد البيانات:** توفِّر بايثون واجهات لجميع قواعد البيانات الأساسية.
- **الرسوميات**: تدعم بايثون التطبيقات الرسوميَّة.
- **دعم البرامج الكبيرة:** بايثون مناسبة للبرامج الكبيرة والمعقَّدة.

3. أين تُستخدَم بايثون؟

تُستخدَم لغة بايثون في كل المجالات، فهي لغة برمجة متعدِّدة الأغراض، ومن مجالات استخدامها: تحليل البيانات، والروبوتات، وتعلم الآلة، وتطبيقات REST، وتطوير المواقع والألعاب، والرسوم ثلاثية الأبعاد، والأتمتة وبرمجة الأنظمة المدمجة، والكثير من المجالات الأخرى التي لا يسعنا حصرها هنا.

تستخدم الكثير من المواقع والشركات العملاقة لغة بايثون، ومنها Spotify و Google وAmazon، إضافة إلى Facebook التي تستخدم بايثون لمعالجة الصور. وفي كل يوم تتحول شركات جديدة إلى استخدام بايثون، مثل Instagram التي قررت مؤخرًا استخدامها وفضلتها على PHP. تُستخدَم بايثون أيضًا من قبل بعض الجهات العلميَّة والبحثيَّة، مثل وكالة الفضاء الأمريكية ناسا، والتي لها مستودع خاص بالمشاريع المُطورة ببايثون.

4. لماذا بايثون وليس غيرها؟

تحديد أفضل لغة برمجة للتعلم قد يكون مهمةً صعبةً. لو سألت عدَّة أشخاص عن لغة البرمجة التي يجب تعلمها، فستحصل على عدة إجابات، ويكفي أن تدخل على جوجل وتكتب أفضل لغة برمجة، وستجد آراءً مختلفةً، وجدالًا لا ينتهي حول هذا الموضوع.

لا أريد أن أبدأ حرب لغات البرمجة هنا، ولكني سأحاول تقديم بعض الحجج لتبرير لماذا أرى أنَّ بايثون هي لغة المستقبل، وأنَّ تعلم لغة بايثون مثالي للمبتدئين الذين يريدون دخول عالم البرمجة وعلوم الحاسوب.

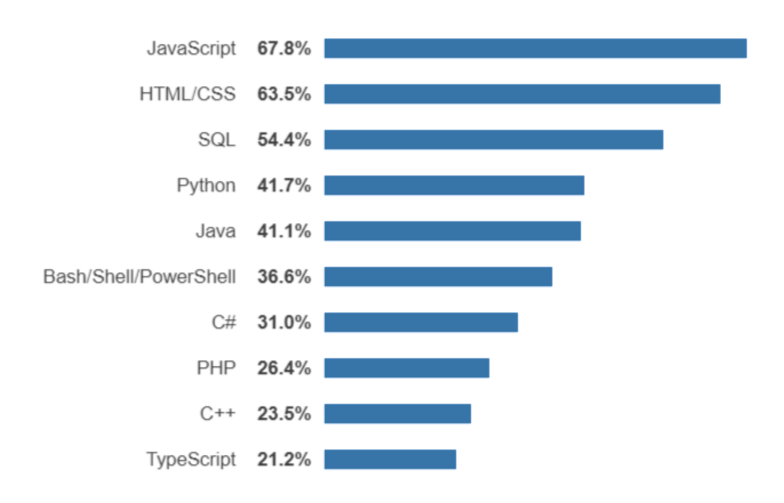
ا. الشعبية

بحسب استطلاع موقع stackoverflow، بايثون هي لغة البرمجة العامَّة الأسرع نموًا، كما تمكنت سنة 2019 من التفوق على جافا بِعدِّها أكثر لغة برمجة متعددة الأغراض استخدامًا، كما أنَّها رابع أكثر لغة تكنولوجيا برمجة استخدامًا وراء JavaScript و HTML/CSS و SQL، كما أنَّها ثاني أكثر لغة برمجة محبوبة من قبل المبرمجين.

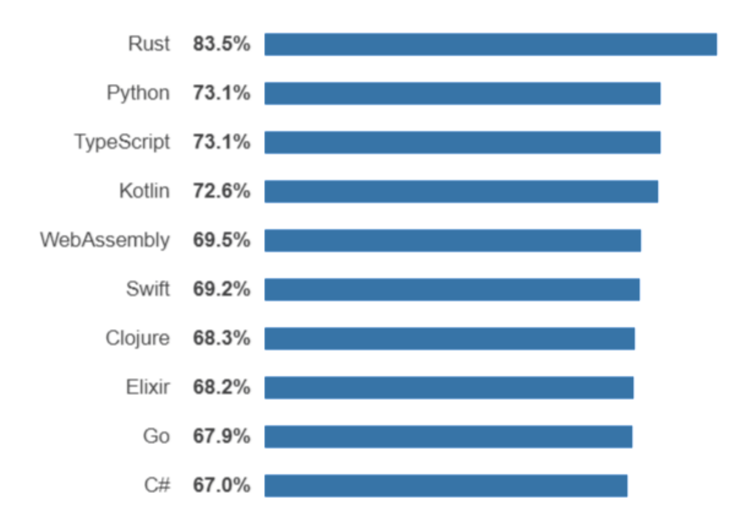
بايثون ليست لغة قويَّة وفعالة وحسب، كما يدل على ذلك حقيقة أنَّها أكثر لغة برمجة متعدِّدة الأغراض استخدامًا، بل هي فوق ذلك محبوبة من قبل المبرمجين، وهذا مؤشر على سهولتها، ثم فوق كل ذلك جميعًا، فإنَّ مستقبلها يبدو مشرقًا، لأنَّها الأسرع نموًا، فهل لا يزال لديك شك في أن تعلم لغة بايثون هو خيار مثالي لك؟!

المخططان البيانيان التاليان يوضحان شعبية لغة بايثون لدى المبرمجين:

- أكثر لغات البرمجة شعبية (المصدر: https://insights.stackoverflow.com/survey/2019)



- أكـــثر لغـــات البرمجـــة المحبوبـــة لـــدى المـــبرمجين (المصـــدر: https://insights.stackoverflow.com/survey/2019)



ب. طلب سوق العمل

يستخدم بايثون بعض أكبر الشركات في مجال التكنولوجيا، مثل PayPal و Uber و Google و Facebook و Instagram و Netflix و Dropbox و Reddit. إضافةً إلى هذا، تُستخدَم بايثون بكثافة في مجال الذكاء الاصطناعي والتعلم الآلي وتحليل البيانات وأنظمة المراقبة وغير ذلك.

يقدر موقع stackoverflow، الدخل السنوي لمطوري بايثون المحترفين بحوالي 63 ألف دولار، وهو مبلغ كبير، ويدل على أنَّ هناك طلبًا كبيرًا على لغة بايثون في سوق العمل.

ج. الدعم

نظرًا لشعبيتها الكبيرة، تتمتع بايثون بدعم جيد على جميع المستويات تقريبًا. وبِعدِّها اللغة المفضلة للمبتدئين، فهناك قناطير من المراجع والمواد والدورات التعليمية التي تشرح مفاهيم البرمجة الأساسيَّة، إضافة إلى صياغة اللغة وتطبيقاتها.

سواء كنت هاويًا، وتحب تعلم البرمجة هوايةً، أو لأجل استخدامها في مجال عملك، مثل تحليل البيانات ومعالجة اللغات الطبيعيَّة وغير ذلك، أو كنت تريد أن تعمل عملًا مستقلًا، أو تدخل سوق العمل وتحقق دخلًا من البرمجة، ففي جميع هذه الحالات، سيكون تعلم لغة بايثون خيارًا مثاليًا لك.

5. خلاصة الفصل

بايثون هي لغة برمجة عالية المستوى، ومُترجمَة (interpreted) وتفاعلية وكائنيَّة، وواحدة من أشهر لغات البرمجة وأكثرها استخدامًا، ويمكن استخدامها في كل المجالات، بدءًا من ألعاب الفيديو ومعالجة اللغات، وحتى تحليل البيانات والتعلُّم الآلي.

أضف إلى ذلك أنها تتمتع بمقروئية عالية، إذ تستخدم كلمات إنجليزية بسيطة، على خلاف اللغات الأخرى التي تستخدم الرموز والكلمات المختصرة، كما أنَّ قواعدها الإملائيَّة والصياغيَّة بسيطة، ما يجعل تعلمها سهلًا موازنةً مع لغات برمجية أخرى. وهذا هو السبب الرئيسي لاعتماد الجامعات ومختلف دورات البرمجة التدريبية تدريسها في البداية لمن يريد الدخول إلى مجال علوم الحاسوب عمومًا والبرمجة خصوصًا.

2

تثبيت بايثون وإعداد بيئة العمل

بعد أن أخذت فكرة عامة عن لغة البرمجة بايثون وتعرفت على تاريخها وإصداراتها - في الفصل السابق - سيرشدك هذا الفصل إلى كيفية تثبيت بايثون على نظام تشغيلك وإعداد البيئة البرمجية اللازمة لكتابة البرامج وتنفيذها خلال رحلتك التعليمية هذه.

الأمور التي سنسلط الضوء عليها هي: تثبيت بايثون 3 ثم إعداد بيئتها البرمجية التي تتمثَّل بإعداد بيئة وهميَّة تمكِّنك من إنشاء مساحة معزولة في حاسوبك مخصصة لمشاريع بايثون، مما يعني أنَّ كل مشروع تعمل عليه يملك مجموعة من الاعتماديات (dependencies) والتي لن تؤثِّر على غيرها من المشاريع. يوفِّر لنا ذلك تحكمًا أكبر بمشاريع بايثون وإمكانية التعامل مع إصداراتٍ مختلفةٍ من حزمها وهذا مهمٌ عندما تتعامل مع الحزم الخارجية. سننشئ بعدئذٍ برنامجًا بسيطًا يعرض العبارة "Hello World!" (أهلًا بالعالم!) الشهيرة، وبهذا سنتحقق من عمل البيئة عملًا صحيحًا، وستصبح آنذاك طريقة إنشاء برامج بايثون وتنفيذها مألوفةً لديك مما يمهِّد الطريق لكتابة وتنفيذ مشاريع بايثون اللاحقة.

اختر مما يلي القسم الخاص بنظام تشغيل حاسوبك (حاولنا شمل أشهر أنظمة التشغيل، لينكس وويندوز وماك) وانتقل إليه لاتباع الخطوات اللازمة لتنفيذ ما سبق.

- ويندوز
- لينكس
  - أوبنتو
  - دبيان
  - سينتوس
- ماك

1. ويندوز

سيُرشِدُك هذا القسم خطوةً بخطوة إلى كيفية تثبيت بايثون 3 في ويندوز 10، وتثبيت بيئة برمجة خاصة بها عبر سطر الأوامر.

ا. المتطلبات المسبقة

يجب أن تملك جهازًا عليه نظام ويندوز 10 متصل بالشبكة مع صلاحيات مدير (administrative access).

ب. فتح وإعداد PowerShell

سنجري معظم أطوار التثبيت والإعدادات عبر سطر الأوامر، والذي هو طريقةٌ غيرُ رسوميةٍ للتعامل مع الحاسوب، فبدلًا من الضغط على الأزرار، ستكتب نصًّا وتعطيه للحاسوب لينفذه، وسيُظهِر لك ناتجًا نصيًا أيضًا. يمكن أن يساعدك سطر الأوامر على تعديل أو أتمتة مختلف المهام التي تنجزها على الحاسوب يوميًا، وهو أداةٌ أساسيةٌ لمطوري البرمجيات.

PowerShell هي برنامج من ميكروسوفت يوفر واجهة سطر الأوامر. يمكن إجراء المهام الإدارية عبر تنفيذ الأصناف cmdlets، والتي تُنطَق "command-lets"، وهي أصناف متخصصة من الإطار NET. يمكنها تنفيذ العمليات. جُعِلت PowerShell مفتوحة المصدر منذ أغسطس 2016، وصارت متوفرة الآن عبر ويندوز وأنظمة يونكس (بما في ذلك ماك ولينكس).

ستعثر على PowerShell بالنقر الأيمن على أيقونة Start في الركن الأيسر السفلي من الشاشة. عندما تنبثق القائمة، انقر على "Search"، ثم اكتب "PowerShell" في شريط البحث. عند تقديم خيارات لك، انقر بالزر الأيمن على تطبيق سطح المكتب "Windows PowerShell".

اختر "Run as Administrator". عندما يظهر مربع حوار يسألك:

> Do you want to allow this app to make changes to your PC?

انقر على "Yes". بمجرد إتمام ذلك، سترى واجهة نصية تبدو كما يلي:

```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> _
```

يمكننا تبديل مجلد النظام عن طريق كتابة الأمر التالي:

```
cd ~
```

بعد ذلك سننتقل إلى المجلد PS C:\Users\Sammy.

لمتابعة عملية التثبيت، سنعدّ بعض الأذونات من خلال PowerShell. تم إعداد PowerShell لتعمل في الوضع الأكثر أمانًا بشكل افتراضي.

هناك عدة مستويات للأذونات، والتي يمكنك إعدادها بوصفك مديرًا (administrator):

- **Restricted**: يمثل سياسة التنفيذ الافتراضية، وبموجب هذا الوضع، لن تتمكن من تنفيذ السكربتات، وستعمل PowerShell بوصفها صدفةً تفاعليةً (أي interactive shell) وحسب.
- **AllSigned**: سيمكِّنك من تنفيذ جميع السكربتات وملفات الإعداد المُوقَّعة من قبل جهة موثوقة، مما يعني أنه من المحتمل أن تُعرِّض جهازك لخطر تنفيذ سكربتات ضارة إن كانت موقَّعة من جهة غير موثوقة.
- **RemoteSigned**: ستمكِّنك من تنفيذ السكربتات وملفات الإعداد المُنزَّلة من الشبكة، والمُوقَّعة من جهة موثوقة، مما يعني احتمال أن تعرِّض جهازك لخطر تنفيذ سكربتات ضارة إن كانت تلك السكربتات الموثوقة ضارة.
- **Unrestricted**: تسمح بتنفيذ جميع السكربتات وملفات الإعداد المُنزَّلة من الشبكة بمجرد تأكيد أنَّك تدرك أنّ الملف مُنزَّلٌ من الشبكة. في هذه الحالة، التوقيعات الرقمية غير لازمة، مما يعني أنَّه من المحتمل تعريض جهازك لخطر تنفيذ سكربتات غير موثوقة منزلة من الشبكة قد تكون ضارة.

سنستخدم سياسة التنفيذ RemoteSigned لتعيين الإذن للمستخدم الحالي، وهكذا سنسمح لبرنامج PowerShell بقبول السكربتات المُنزَّلة التي نثق بها، ودون خفض كل دفاعاتنا وجعل الأذونات هشَّة كما هو الحال مع سياسة التنفيذ Unrestricted. سنكتب في PowerShell:

```
Set-ExecutionPolicy -Scope CurrentUser
```

ستطالبك PowerShell بتحديد سياسة التنفيذ، وبما أنَّنا نريد استخدام RemoteSigned، فسنكتب:

```
RemoteSigned
```

بمجرد الضغط على الزر الإدخال (enter)، ستُسأل عما إن كنت نريد تغيير سياسة التنفيذ. اكتب الحرف y لاختيار "نعم"، واعتماد التغييرات. يمكننا التحقق من نجاح العملية عن طريق طلب الأذونات الحالية في الجهاز عبر كتابة:

```
Get-ExecutionPolicy -List
```

ستحصل على مخرجات مشابهة لما يلي:

```
        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser    RemoteSigned
 LocalMachine       Undefined
```

هذا يؤكد أنَّ المستخدم الحالي يمكنه تنفيذ السكربتات الموثوقة التي نُزِّلت من الشبكة. يمكننا الآن تنزيل الملفات التي سنحتاج إليها لإعداد بيئة برمجة بايثون.

ج. تثبيت Chocolatey

مدير الحزم (package manager) هو مجموعة من أدوات البرمجيات التي تعمل على أتمتة عمليات التثبيت، بما في ذلك التثبيت الأولي للبرامج، وترقيتها، وإعدادها، وإزالتها عند الحاجة.

تحفظ هذه الأدوات التثبيتات في موقع مركزي، ويمكنها صيانة جميع حزم البرامج على النظام وفق تنسيقات (formats) معروفة.

يعد Chocolatey مدير حزم مفتوح المصدر يعمل من سطر الأوامر، صمِّم لنظام ويندوز،

وتحاكي apt-get الخاص بلينكس، ويمكنه مساعدتك على تثبيت التطبيقات والأدوات بسرعة. سنستخدمه لتنزيل ما نحتاج إليه لبيئتنا التطويرية.

قبل تثبيت السكربت، دعنا نقرؤه للتأكد من أنَّ التغييرات التي سيجريها على الجهاز مقبولة. سنستخدم إطار العمل NET. لتنزيل وعرض السكربت Chocolatey في نافذة الطرفية. سننشئ كائنًا WebClient يُسمى $script (يمكنك تسميته كما تريد طالما ستستخدم المحرف $ في البداية)، والذي يشارك إعدادات الاتصال بالشبكة مع المتصفح Internet Explorer:

```
$script = New-Object Net.WebClient
```

دعنا نلقي نظرة على الخيارات المتاحة لنا من خلال توصيل الكائن إلى الصنف Get-Member لإعادة جميع الأعضاء (الخاصيات والتوابع) الخاصة بكائن WebClient:

```
$script | Get-Member
```

سنحصل على المخرجات التالية:

```
. . .
DownloadFileAsync      Method    void DownloadFileAsync(uri
address, string fileName), void DownloadFileAsync(ur...
DownloadFileTaskAsync    Method    System.Threading.Tasks.Task
DownloadFileTaskAsync(string address, string fileNa...
DownloadString         Method    string DownloadString(string
address), string DownloadString(uri address) # هذا التابع
DownloadStringAsync     Method    void DownloadStringAsync(uri
address), void DownloadStringAsync(uri address, Sy...
DownloadStringTaskAsync  Method
System.Threading.Tasks.Task[string]
DownloadStringTaskAsync(string address), Sy…
 . . .
```

عند النظر إلى المخرجات، يمكننا تحديد التابع DownloadString الذي يمكننا استخدامه لعرض محتوى السكربت والتوقيع في نافذة PowerShell كما يلي:

```
$script.DownloadString("https://chocolatey.org/install.ps1")
```

بعد مطالعة السكربت، يمكننا تثبيت Chocolatey عن طريق كتابة ما يلي في PowerShell:

```
iwr https://chocolatey.org/install.ps1 -UseBasicParsing | iex
```

تسمح لنا iwr أو Invoke-WebRequest التي تخص cmdlet باستخراج البيانات من الشبكة. سيؤدي هذا إلى تمرير السكربت إلى iex أو Invoke-Expression، والذي سينفذ محتويات السكربت، وتنفيذ سكربت التثبيت لمدير الحزم Chocolatey.

اسمح لبرنامج PowerShell بتثبيت Chocolatey. بمجرد تثبيته بالكامل، يمكننا البدء في تثبيت أدوات إضافية باستخدام الأمر choco.

إن احتجت إلى ترقية Chocolatey مستقبلًا، يمكنك تنفيذ الأمر التالي:

```
choco upgrade chocolatey
```

بعد تثبيت مدير الحزم، يمكننا متابعة تثبيت ما نحتاجه لبيئة البرمجة خاصتنا.

د. تثبيت محرر النصوص nano (اختياري)

سنثبّت الآن nano، وهو محرِّر نصوص يستخدم واجهة سطر الأوامر، والذي يمكننا استخدامه لكتابة البرامج مباشرة داخل PowerShell. هذه ليست خطوة إلزامية، إذ يمكنك بدلا من ذلك استخدام محرر نصوص بواجهة مستخدم رسومية مثل Notepad، لكن ميزة nano أنه سيُعوِّدك على استخدام PowerShell. دعنا نستخدم Chocolatey لتثبيت nano وذلك بتنفيذ الأمر التالي:

```
choco install -y nano
```

الخيار y- يعني أنَّك توافق على تنفيذ السكربت تلقائيًا دون الحاجة إلى تأكيد. بعد تثبيت nano، سنكون قادرين على استخدام الأمر nano لإنشاء ملفات نصية جديدة، وسنستخدمه بعد حين لكتابة أول برامجنا في بايثون.

ه. تثبيت بايثون 3

مثلما فعلنا مع nano أعلاه، سنستخدم Chocolatey لتثبيت بايثون 3:

```
choco install -y python3
```

ستثبِّت PowerShell الآن بايثون 3، مع عرض بعض المخرجات أثناء العملية. بعد اكتمال العملية، سترى المخرجات التالية:

```
Environment Vars (like PATH) have changed. Close/reopen your
shell to
 See the changes (or in powershell/cmd.exe just type
'refreshenv').
The install of python3 was successful.

 Software installed as 'EXE', install location is    likely
default.

Chocolatey installed 1/1 packages. 0 packages failed.
 See the log for details (C:\ProgramData\chocolatey\logs\
chocolatey.log).
```

بعد الانتهاء من التثبيت، ستحتاج إلى التحقق من تثبيت بايثون وجهوزيتها للعمل. لرؤية التغييرات، استخدم الأمر refreshenv أو أغلق PowerShell ثم أعد فتحها بصلاحيات مدير

النظام، ثم تحقق من إصدار بايثون على جهازك:

```
python -V
```

ستحصل على مخرجات في نافذة الطرفية والتي ستريك إصدار بايثون المثبَّت.

```
Python 3.7.0
```

سنثبِّت الأداة pip، إلى جانب بايثون، وهي أداةٌ تعمل مع لغة بايثون تُثَبِّت وتدير الحزم البرمجية التي قد نحتاج إلى استخدامها في تطوير مشاريعنا (سنتعلم المزيد عن الوحدات والحزم التي يمكنك تثبيتها بالأداة pip في الفصل المخصَّص للوحدات).

سنحدِّث pip عبر الأمر التالي:

```
python -m pip install --upgrade pip
```

يمكننا استدعاء بايثون من Chocolatey عبر الأمر python. سنستخدم الراية -m لتنفيذ الوحدة كأنها سكربت، وإنهاء قائمة الخيارات، ومن ثمَّ نستخدم pip لتثبيت الإصدار الأحدث.

بعد تثبيت بايثون وتحديث pip، فنحن جاهزون لإعداد بيئة افتراضية لمشاريع التطوير خاصتنا.

و. إعداد بيئة افتراضية

الآن بعد تثبيت Chocolatey و nano وبايثون، يمكننا المضي قدمًا لإنشاء بيئة البرمجة خاصتنا عبر الوحدة venv.

تُمكِّنك البيئات الافتراضية من إنشاء مساحة معزولة في حاسوبك مخصصة لمشاريع بايثون، مما يعني أنَّ كل مشروع تعمل عليه ستكون له اعتماديَّاته (dependencies) الخاصة به، والتي لن تؤثِّر على غيره من المشاريع.

يوفِّر لنا ضبط بيئةٍ برمجيةٍ تحكمًا أكبر بمشاريع بايثون، وإمكانية التعامل مع إصداراتٍ مختلفةٍ من حزم بايثون. وهذا مهمٌ كثيرًا عندما تتعامل مع الحزم الخارجية.

يمكنك ضبط أيِّ عددٍ تشاء من البيئات الافتراضيَّة، وكل بيئة ستكون ممثلة بمجلد في حاسوبك يحتوي على عدد من السكربتات.

اختر المجلد الذي تريد أن تضع فيه بيئات بايثون، أو يمكنك إنشاء مجلد جديد باستخدام الأمر `mkdir` كما يلي:

```
mkdir environments
cd environments
```

بعد أن انتقلتَ إلى المجلد الذي تريد احتواء البيئات فيه، تستطيع الآن إنشاء بيئة جديدة بتنفيذ الأمر التالي:

```
python -m venv my_env
```

سننفِّذ باستخدام الأمر `python` الوحدة `venv` لإنشاء البيئة الافتراضية التي أطلقنا عليها في هذه الحالة `my_env`.

ستنشئ `venv` مجلدًا جديدًا يحتوي على بعض العناصر التي يمكن عرضها باستخدام الأمر `ls`:

```
ls my_env
```

سنحصل على المخرجات التالية:

```
Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        8/22/2016    2:20 PM               Include
```

```
d-----        8/22/2016   2:20 PM                Lib
d-----        8/22/2016   2:20 PM                Scripts
-a----        8/22/2016   2:20 PM            107 pyvenv.cfg
```

تعمل هذه الملفات مع بعضها لضمان أن تكون مشاريعك معزولةً عن سياق الآلة المحلية، لكي لا تختلط ملفات النظام مع ملفات المشاريع. وهذا أمرٌ حسنٌ لإدارة الإصدارات ولضمان أنَّ كل مشروع يملك وصولًا إلى الحزمِ التي يحتاجها.

عليك تفعيل البيئة لاستخدامها، وذلك بكتابة الأمر التالي الذي سيُنفِّذ سكربت التفعيل في المجلد `Scripts`:

```
my_env\Scripts\activate
```

يجب أن تظهر الآن سابقةٌ (prefix) في المِحَث (prompt) والتي هي اسم البيئة المستخدمة، وفي حالتنا هذه يكون اسمها `my_env`.

```
(my_env) PS C:\Users\Sammy\Environments>
```

تتيح لنا هذه البادئة معرفة أنَّ البيئة `my_env` مفعَّلة حاليًا، وهذا يعني أنَّنا لن سنستخدم إلا إعدادات وحزم هذه البيئة عند إنشاء مشاريع جديدة.

ز. إنشاء برنامج بسيط

بعد أن أكملنا ضبط بيئتنا الافتراضية، لننشِئ برنامجا بسيطا يعرض العبارة «مرحبا بالعالم!»، وبهذا سنتحقق من أنَّ البيئة تعمل بالشكل الصحيح، ولكي تتعوَّد على إنشاء برامج بايثون إن كنتَ وافدًا جديدًا على اللغة.

علينا أولًا تشغيل المحرر nano وإنشاء ملف جديد:

```
(my_env) PS C:\Users\Sammy> nano hello.py
```

بعد فتح الملف في نافذة الطرفية، سنكتب البرنامج الخاص بنا:

```
print("Hello, World!")
```

أغلق محرر nano بالضغط على Ctrl+x ثم اضغط على y عندما يسألك عن حفظ الملف.

بعد أن يُغلَق المُحرِّر nano وتعود إلى سطر الأوامر، حاول تنفيذ البرنامج:

```
(my_env) PS C:\Users\Sammy> python hello.py
```

سيؤدي برنامج `hello.py` الذي أنشأتَه إلى طباعة الناتج التالي في الطرفية:

```
Hello, World!
```

للخروج من البيئة، اكتب الأمر `deactivate` وستعود إلى مجلدك الأصلي. حان الآن الوقت للتعمق بلغة بايثون وإنشاء برامج رائعة! انتقل إلى الفصل التالي، استخدام سطر أوامر بايثون التفاعلي.

2. أوبنتو

سيُرشِدُك هذا القسم خطوةً بخطوة إلى كيفية تثبيت بايثون 3 على خادم أوبنتو 20.04، البرمجة على الخوادم لها العديد من الميزات، كما تدعم المشاريع التعاونية.

صحيحٌ أنَّ هذا القسم يشرح عملية التثبيت على خادم أوبنتو 20.04، إلا أنَّ المفاهيم الأساسية فيه تنطبق على جميع توزيعات دبيان لينكس (Debian Linux).

ا. المتطلبات المسبقة

يجب أن تملك صلاحيات مستخدم غير جذري (non-root user) مع امتيازات sudo على خادم أوبنتو 20.04. إذا لم تكن لك خبرة في التعامل مع بيئة النافذة الطرفية، فيمكنك مطالعة المقال «مدخل إلى طرفيّة لينكس Linux Terminal».

ب. إعداد بايثون 3

في أوبنتو 20.04 والإصدارات الأخرى من دبيان لينكس، ستجد أن بايثون 3 مثبَّتة مسبقًا. للتأكد من أنَّ إصدارات بايثون حديثة، سنحدّث النظام ونرقِّيه باستخدام الأمر apt للعمل مع أداة التحزيم المتقدمة من أوبنتو (Ubuntu’s Advanced Packaging Tool):

```
sudo apt update
sudo apt -y upgrade
```

الخيار y- يعني أنَّك توافق على تثبيت جميع الحزم القابلة للتحديث، لكن قد تحتاج إلى تأكيد ذلك عند تحديث النظام وذلك اعتمادًا على الحزم التي ستُحدَّث، ونسخة نظامك. بعد إكمال العملية، يمكن التحقق من إصدار بايثون 3 المُثبَّت في النظام بكتابة:

```
python3 -V
```

ستحصل على مخرجات في نافذة الطرفية والتي ستريك إصدار بايثون المثبَّت. قد يختلف الرقم بناءً على النسخة المثبتة في توزيعتك، لكن يجب أن يكون شبيهًا بما يلي:

```
Python 3.8.2
```

لإدارة الحزم البرمجية الخاصة ببايثون، سنثبّت pip، وهي أداةٌ تعمل مع لغة بايثون تُثَبِّت وتدير الحزم البرمجية التي قد نحتاج إلى استخدامها في تطوير مشاريعنا (سنتعلم المزيد عن الوحدات والحزم التي يمكنك تثبيتها بالأداة pip في الفصل المخصَّص للوحدات):

```
sudo apt install -y python3-pip
```

يمكن تثبيت حزم بايثون بكتابة ما يلي مع تبديل package_name باسم الحزمة:

```
pip3 install package_name
```

عليك وضع اسم الحزمة أو المكتبة التابعة لبايثون مكان package_name مثل Django لتطوير الويب، أو NumPy لتثبيتها؛ لذا، إن شئتَ تثبيت NumPy فيمكنك تنفيذ الأمر pip3 install numpy.

هناك عدة حزم وأدوات تطوير أخرى يجب تثبيتها للتأكد من أنّ بيئة البرمجة جاهزة:

```
sudo apt install -y build-essential libssl-dev libffi-dev
python3-dev
```

بعد أن انتهينا من ضبط بايثون وتثبيت pip، يمكننا الآن إنشاء «بيئة افتراضية» (virtual environment) لمشاريعنا.

ج. إعداد بيئة افتراضية

تُمكّنك البيئات الافتراضية من إنشاء مساحة معزولة في خادمك مخصصة لمشاريع بايثون، مما يعني أنَّ كل مشروع تعمل عليه ستكون له اعتماديَّاته (dependencies) الخاصة به، والتي لن تؤثِّر على غيره من المشاريع.

يوفِّر لنا ضبط بيئةٍ برمجيةٍ تحكمًا أكبر بمشاريع بايثون، وإمكانية التعامل مع إصداراتٍ مختلفةٍ من حزم بايثون. وهذا مهمٌ كثيرًا عندما تتعامل مع الحزم الخارجية.

يمكنك ضبط أيِّ عددٍ تشاء من البيئات الافتراضية، وكل بيئة ستكون ممثلة بمجلد في خادمك يحتوي على عدد من السكربتات.

هناك عدة طرق لإعداد بيئة برمجية في بايثون، لكننا سنستخدم وحدة (module) برمجية باسم venv، وهي جزءٌ من مكتبة بايثون 3 القياسية. سنثبِّت venv على نظامنا بتنفيذ الأمر التالي:

```
sudo apt-get install -y python3-venv
```

بعد إتمام التثبيت، فنحن جاهزون لإنشاء البيئات الافتراضية، يمكننا الآن إمَّا اختيار مجلد نضع فيه بيئات بايثون، أو إنشاء مجلد جديد باستخدام الأمر `mkdir` كما يلي:

```
mkdir environments
cd environments
```

بعد أن انتقلتَ إلى المجلد الذي تريد احتواء البيئات فيه، تستطيع الآن إنشاء بيئة جديدة بتنفيذ الأمر الآتي:

```
python3.6 -m venv my_env
```

سيُنشِئ الأمر pyvenv مجلدًا جديدًا فيه بعض الملفات التي يمكن عرضها باستخدام الأمر ls:

```
ls my_env
```

ستظهر لك مخرجات شبيهة بالمخرجات التالية:

```
bin include lib lib64 pyvenv.cfg share
```

تعمل هذه الملفات مع بعضها لضمان أنَ تكون مشاريعك معزولةٌ عن سياق الآلة المحلية، لكي لا تختلط ملفات النظام مع ملفات المشاريع. وهذا أمرٌ حسنٌ لإدارة الإصدارات ولضمان أنَّ كل مشروع يملك وصولًا إلى الحزمٍ التي يحتاج إليها. تتوافر أيضًا Python Wheels، والتي هي صيغة حزمٍ مبنية (built-package format) لبايثون، والتي يمكن أن تُسرِّع من تطوير البرامج بتقليل عدد المرات التي تحتاج فيها إلى تصريف (compile) المشروع، وهي موجودةٌ في المجلد share في توزيعة أوبنتو 20.04.

عليك تفعيل البيئة لاستخدامها، وذلك بكتابة الأمر التالي الذي سيُنفِّذ سكربت التفعيل:

```
source my_env/bin/activate
```

يجب أن تظهر الآن سابقةٌ (prefix) في المِحث (prompt) والتي هي اسم البيئة المستخدمة، وفي حالتنا هذه يكون اسمها my_env، وقد يكون مظهر المِحَث مختلفًا في توزيعة دبيان، وذلك اعتمادًا على الإصدار المستخدم؛ لكن يجب أن تشاهد اسم البيئة بين قوسين في بداية السطر:

```
(my_env) sammy@ubuntu:~/environments$
```

ستسمح لك السابقة بمعرفة أنَّ البيئة my_env مفعَّلة حاليا، وهذا يعني أنَّنا سنستخدم إعدادات وحزم هذه البيئة عند إنشاء مشاريع جديدة.

يجب أن تكون بيئتك الافتراضية جاهزةً للاستخدام بعد اتباعك للخطوات السابقة.

ملاحظة: يمكنك داخل البيئة الافتراضية أن تستخدم الأمر python بدلًا من python3 والأمر pip بدلًا من pip3 إن شئتَ أما إذا كنتَ تستخدم بايثون3 خارج البيئة الافتراضيَّة، فيجب عليك حينها استخدام python3 و pip3 حصرًا.

د. إنشاء برنامج بسيط

بعد أن أكملنا ضبط بيئتنا الافتراضية، لننشِئ برنامجا بسيطا يعرض العبارة «Hello World!»، وبهذا سنَتحقَّق من أنَّ البيئة تعمل بالشكل الصحيح، ولكي تتعوَّد على إنشاء برامج بايثون إن كنتَ وافدًا جديدًا على اللغة.

علينا أولًا تشغيل محرر ملفات نصية لإنشاء ملف جديد، وليكن المُحرِّر nano الذي يعمل من سطر الأوامر:

```
(my_env) sammy@ubuntu:~/environments$ nano hello.py
```

بعد فتح الملف في نافذة الطرفية، سنكتب البرنامج الخاص بنا:

```
print("Hello World!")
```

أغلق محرر nano بالضغط على Ctrl+x ثم اضغط على y عندما يسألك عن حفظ الملف. بعد أن يُغلَق المحرر nano وتعود إلى سطر الأوامر، حاول تنفيذ البرنامج:

```
(my_env) sammy@ubuntu:~/environments$ python hello.py
```

سيؤدي برنامج `hello.py` الذي أنشأتَه إلى طباعة الناتج الآتي في الطرفية:

```
Hello World!
```

للخروج من البيئة، اكتب الأمر `deactivate` وستعود إلى مجلدك الأصلي.

تهانينا! لقد ضبطتَ الآن بيئة تطوير للغة بايثون 3 على خادم أوبنتو، وحان الوقت للتعمق بلغة بايثون وإنشاء برامج رائعة! انتقل إلى الفصل التالي، استخدام سطر أوامر بايثون التفاعلي.

3. دبيان

سيُرشِدُك هذا القسم خطوةً بخطوة إلى كيفية تثبيت بايثون 3 على لينكس، وتثبيت بيئة برمجة عبر سطر الأوامر. صحيحٌ أنَّ هذا القسم يشرح عملية التثبيت في دبيان 10، إلا أنَّ المفاهيم الأساسية فيه تنطبق على جميع توزيعات دبيان لينكس (Debian Linux).

ا. المتطلبات المسبقة

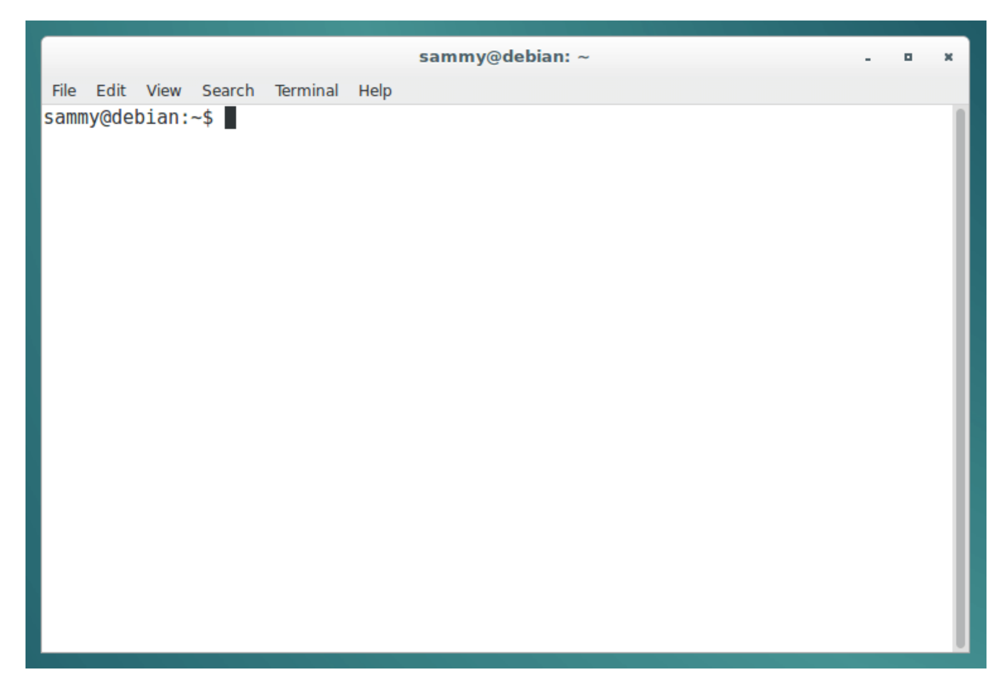
يجب أن تملك صلاحيات مستخدم غير جذري (non-root user) مع امتيازات sudo على توزيعة دبيان 10، أو توزيعة أخرى من دبيان لينكس (Debian Linux).

إذا لم تكن لك خبرة في التعامل مع بيئة النافذة الطرفية، فيمكنك مطالعة المقال «مدخل إلى طرفيّة لينكس Linux Terminal».

ب. إعداد بايثون 3

ستُثبّت ونضبط بايثون عبر سطر الأوامر، والذي هو طريقةٌ غيرُ رسوميةٍ للتعامل مع الحاسوب، فبدلًا من الضغط على الأزرار، ستكتب نصًّا وتعطيه للحاسوب لينفِّذه، وسيُظهِر لك ناتجًا نصيًا أيضًا. يمكن أن يساعدك سطر الأوامر على تعديل أو أتمتة مختلف المهام التي تنجزها على الحاسوب يوميا، وهو أداةٌ أساسيةٌ لمطوري البرمجيات، وهنالك الكثير من الأوامر التي عليك تعلمها لكي تتمكن من الاستفادة منه. هنالك مقالات في أكاديمية حسوب (مثل مقال مدخل إلى طرفية لينكس Linux Terminal) ستعلمك أساسيات سطر الأوامر، وهنالك كتاب «سطر أوامر لينكس» الذي يُعدُّ مرجعًا لطريقة التعامل مع سطر الأوامر.

ستجد تطبيق Terminal (البرنامج الذي تستعمله للوصول إلى سطر الأوامر) بفتح القائمة في الزاوية السفلى اليسرى من الشاشة ثم كتابة «terminal» في شريط البحث، ثم الضغط على أيقونة التطبيق التي ستظهر بعدئذٍ أو يمكنك أن تضغط على Ctrl+Alt+T في لوحة المفاتيح بنفس الوقت لتشغيل تطبيق Terminal.



في دبيان 10 والإصدارات الأخرى من دبيان لينكس، ستجد بايثون مثبَّتة مسبقًا. للتأكد من أنَّ إصدارات بايثون حديثة، سنحدِّث النظام ونرقِّيه باستخدام الأمر `apt`:

```
sudo apt update
sudo apt -y upgrade
```

الخيار y- يعني أنَّك توافق على تثبيت جميع الحزم القابلة للتحديث، لكن قد تحتاج إلى تأكيد ذلك عند تحديث النظام وذلك اعتمادًا على الحزم التي ستُحدَّث، ونسخة لينكس.

بعد إكمال العمليَّة، يمكننا التحقق من إصدار بايثون 3 المُثبَّت في النظام بكتابة:

```
python3 -V
```

ستحصل على مخرجات في نافذة الطرفية والتي ستريك إصدار بايثون المثبَّت. قد يختلف الرقم بناءً على النسخة المثبّتة في توزيعتك، لكن يجب أن يكون شبيهًا بما يلي:

```
Python 3.7.3
```

لإدارة الحزم البرمجية الخاصة ببايثون، سنثبّت pip، وهي أداةٌ تعمل مع لغة بايثون تُثبِّت وتدير الحزم البرمجية التي قد نحتاج إلى استخدامها في تطوير مشاريعنا (سنتعلم المزيد عن الوحدات والحزم التي يمكنك تثبيتها بالأداة pip في الفصل المخصَّص للوحدات):

```
sudo apt install -y python3-pip
```

يمكن تثبيت حزم بايثون بكتابة ما يلي مع تبديل package_name باسم الحزمة:

```
pip3 install package_name
```

عليك وضع اسم الحزمة أو المكتبة التابعة لبايثون مكان package_name مثل Django لتطوير الويب، أو NumPy لإجراء الحسابات العلمية؛ لذا، إن شئتَ تنزيل NumPy فيمكنك تنفيذ الأمر pip3 install numpy.

هناك عدة حزم وأدوات تطوير أخرى يجب تثبيتها للتأكد من أنّ بيئة البرمجة جاهزة:

```
sudo apt install build-essential libssl-dev libffi-dev python3-
dev
```

بعـد أن انتهينـا من ضـبط بـايثون وتثـبيت pip، يمكننـا الآن إنشـاء «بيئـة افتراضـية» (virtual environment) لمشاريعنا.

ج. إعداد بيئة افتراضية

تُمكِّنـك البيئـات الافتراضـية من إنشـاء مسـاحة معزولـة في حاسـوبك مخصصـة لمشـاريع بايثون، مما يعني أنَّ كل مشروع تعمل عليه ستكون له اعتماديَّاته (dependencies) الخاصة به، والتي لن تؤثِّر على غيره من المشاريع.

يـوفِّر لنا ضبط بيئـةٍ برمجيـةٍ تحكما أكبر بمشـاريع بـايثون، وإمكانيَّة التعامل مع إصداراتٍ مختلفةٍ من حزم بايثون. وهذا مهمٌ كثيرًا عندما تتعامل مع الحزم الخارجية.

يمكنك ضبط أيِّ عـددٍ تشـاء من البيئـات الافتراضـية، وكـل بيئـة سـتكون ممثلـة بمجلـد في حاسوبك يحتوي على عدد من السكربتات.

هناك عدة طرق لإعداد بيئة برمجية في بايثون، لكننا سنستخدم وحدة (module) برمجية باسم venv، وهي جزءٌ من مكتبة بايثون 3 القياسية. سنثبِّت venv على نظامنا بكتابة:

```
sudo apt install -y python3-venv
```

بعد إتمام التثبيت، فنحن جاهزون لإنشاء البيئـات الافتراضـية، يمكننا الآن إمَّا اختيـار مجلد نضع فيه بيئات بايثون، أو إنشاء مجلد جديد باستخدام الأمر `mkdir` كما يلي:

```
mkdir environments
cd environments
```

بعد أن انتقلتَ إلى المجلد الذي تريد احتـواء البيئـات فيـه، تسـتطيع الآن إنشـاء بيئـة جديـدة بتنفيذ الأمر الآتي:

```
python3.7 -m venv my_env
```

سيُنشِئ الأمر pyvenv مجلدًا جديدًا فيه بعض الملفات التي يمكن عرضها باستخدام الأمر ls:

```
ls my_env
```

ستظهر لك مخرجات شبيهة بالمخرجات التالية:

```
bin include lib lib64 pyvenv.cfg share
```

تعمل هذه الملفات مع بعضها لضمان أنَ تكون مشاريعك معزولةٌ عن سياق الآلة المحلية، لكي لا تختلط ملفات النظام مع ملفات المشاريع. وهذا أمرٌ حسنٌ لإدارة الإصدارات ولضمان أنَّ كل مشروع يملك وصولًا إلى الحزمٍ التي يحتاج إليها. تتوافر أيضًا Python Wheels، والتي هي صيغة بناء حزمٍ (مثل built-package format) لبايثون، والتي يمكن أن تُسرِّع من تطوير البرامج بتقليل عدد المرات التي تحتاج فيها إلى تصريف (compile) المشروع، وهي موجودةٌ في كل المجلدات المُسمّاة lib.

عليك تفعيل البيئة لاستخدامها، وذلك بكتابة الأمر التالي الذي سيُنفِّذ سكربت التفعيل:

```
source my_env/bin/activate
```

يجب أن تظهر الآن سابقةٌ (prefix) في المِحث (prompt) والتي هي اسم البيئة المستخدمة، وفي حالتنا هذه يكون اسمها my_env، وقد يكون مظهر المِحَث مختلفًا في توزيعة دبيان، وذلك اعتمادًا على الإصدار المستخدم؛ لكن يجب أن تشاهد اسم البيئة بين قوسين في بداية السطر:

```
(my_env) sammy@sammy:~/environments$
```

ستسمح لك السابقة بمعرفة أنّ البيئة my_env مفعلة حاليا، وهذا يعني أننا سنستخدم إعدادات وحزم هذه البيئة عند إنشاء مشاريع جديدة.

يجب أن تكون بيئتك الافتراضية جاهزةً للاستخدام بعد اتباعك للخطوات السابقة.

ملاحظة: يمكنك داخل البيئة الافتراضية أن تستخدم الأمر python بدلًا من python3 والأمر pip بدلًا من pip3 إن شئتَ أما إذا كنتَ تستخدم بايثون3 خارج البيئة الافتراضية، فيجب عليك حينها استخدام python3 و pip3 حصرًا.

د. إنشاء برنامج بسيط

بعد إكمال ضبط بيئتنا الافتراضيّة، لننشِئ برنامجًا بسيطًا يعرض العبارة «مرحبا بالعالم!»، وبهذا سنتحقق من أنَّ البيئة تعمل بالشكل الصحيح، ولكي تتعوّد على إنشاء برامج بايثون إن كنتَ وافدًا جديدًا على اللغة.

علينا أولًا تشغيل محرر ملفات نصية لإنشاء ملف جديد، وليكن المحرر nano الذي يعمل من سطر الأوامر:

```
(my_env) sammy@sammy:~/environments$ nano hello.py
```

بعد فتح الملف في نافذة الطرفية، سنكتب البرنامج الخاص بنا:

```
print("Hello, World!")
```

أغلق محرر nano بالضغط على Ctrl+x ثم اضغط على y عندما يسألك عن حفظ الملف.

بعد أن يُغلَق المحرر nano وتعود إلى سطر الأوامر، حاول تنفيذ البرنامج:

```
(my_env) sammy@sammy:~/environments$ python hello.py
```

سيؤدي برنامج `hello.py` الذي أنشأتَه إلى طباعة الناتج التالي في الطرفية:

```
Hello, World!
```

للخروج من البيئة، اكتب الأمر `deactivate` وستعود إلى مجلدك الأصلي.

تهانينا! لقد ضبطتَ الآن بيئة تطويرٍ للغة بايثون 3 في نظام لينكس دبيان، حان الآن الوقت للتعمق بلغة بايثون وإنشاء برامج رائعة! انتقل إلى الفصل التالي، استخدام سطر أوامر بايثون التفاعلي.

4. CentOS

سيُرشِدُك هذا القسم خطوةً بخطوة إلى كيفية تثبيت بايثون 3 على CentOS 8، وتثبيت بيئة برمجة عبر سطر الأوامر.

ا. المتطلبات المسبقة

يجب أن تملك صلاحيات مستخدم أساسي غير جذري (non-root superuser) على نظام CentOS 8 متصل بالشبكة.

إذا لم تكن لك خبرة في التعامل مع بيئة النافذة الطرفية، فيمكنك مطالعة المقال «مدخل إلى طرفيّة لينكس Linux Terminal».

ب. تحضير النظام

سنُثبِّت ونضبط بايثون عبر سطر الأوامر، إن كان نظام CentOS 8 يبدأ بسطح مكتب ذي واجهة مستخدم رسومية (GUI)، فيمكنك الدخول إلى سطر الأوامر بفتح القائمة، والدخول إلى Applications ثم Utilities ثم النقر على Terminal. هنالك مقالات في أكاديمية حسوب (مثل مقال مدخل إلى طرفيّة لينكس Linux Terminal) ستعلمك أساسيات سطر الأوامر، وهنالك كتاب «سطر أوامر لينكس» الذي يُعدُّ مرجعًا لطريقة التعامل مع سطر الأوامر.

سنستخدم أداة إدارة الحزم مفتوحة المصدر DNF (اختصار للعبارة Dandified YUM وهو الجيل الثاني من مدير الحزم Yellowdog Updater, Modified المعروف بالاختصار YUM). هذه أداة شائعة الاستخدام لإدارة الحزم على أنظمة لينكس المستندة إلى Red Hat، مثل CentOS، إذ تتيح لك تثبيت الحزم وتحديثها بسهولة، وكذلك إزالة الحزم من الجهاز.

قبل أن نبدأ التثبيت، دعنا نتأكد من أنَّ لدينا أحدث إصدار من yum عبر تنفيذ الأمر التالي:

```
sudo dnf update -y
```

الخيار y- يعني أنَّك تدرك أنَّك تحدث تغييرات، وذلك لمنع الطرفيَّة من طلب تأكيد قبل تنفيذ الأمر.

بمجرَّد تثبيت وتحديث كل شيء، فنحن جاهزون لتثبيت بايثون 3.

ج. تثبيت وإعداد بايثون 3

نظام CentOS مشتق من RHEL (اختصار للجملة Red Hat Enterprise Linux)، والذي يركز على الثبات والاستقرار. أي لن تجد في هذا النظام إلا الإصدارات المستقرة والمُختبرة من التطبيقات والحزم القابلة للتنزيل، لذلك وباستعمال مدير حزم CentOS ستجد أحدث إصدار من بايثون دومًا:

```
sudo dnf install python3 -y
```

بعد إكمال العملية، يمكننا التحقق من نجاح عملية التثبيت بطلب إصدار بايثون بكتابة:

```
python3 -V
```

ستحصل على مخرجات في نافذة الطرفية والتي ستريك إصدار بايثون المثبَّت. قد يختلف الرقم بناءً على النسخة المثبتة في توزيعتك، لكن يجب أن يكون شبيهًا بما يلي:

```
Python 3.6.8
```

سنثبّت تاليًا أدوات تطوير CentOS التي تسمح لك ببناء التطبيقات وتصريفها انطلاقًا من شيفرتها المصدرية:

```
sudo dnf -y groupinstall development
```

ننتقل بعد اكتمال التثبيت للخطوة التالية وهي ضبط وإعداد بيئة تطويرية لكتابة برامج بايثون وتنفيذها.

د. إعداد بيئة افتراضية

الآن بعد أن ثبّتنا بايثون وأعددنا النظام، يمكننا المضي قدمًا لإنشاء بيئة البرمجة التي سنعمل فيها باستخدام venv.

تُمكّنك البيئات الافتراضية من إنشاء مساحة معزولة في حاسوبك مخصّصة لمشاريع بايثون، مما يعني أنّ كل مشروع تعمل عليه ستكون له اعتماديّاته (dependencies) الخاصة به، والتي لن تؤثّر على غيره من المشاريع.

يوفّر لنا ضبط بيئةٍ برمجيةٍ تحكمًا أكبر بمشاريع بايثون، وإمكانيّة التعامل مع إصداراتٍ مختلفةٍ من حزم بايثون. وهذا مهمٌ كثيرًا عندما تتعامل مع الحزم الخارجية.

يمكنك ضبط أيّ عددٍ تشاء من البيئات الافتراضيّة، وكل بيئة ستكون ممثلة بمجلد في حاسوبك يحتوي على عدد من السكربتات.

بعد إتمام التثبيت، فنحن جاهزون لإنشاء البيئات الافتراضية، يمكننا الآن إما اختيار مجلد نضع فيه بيئات بايثون، أو إنشاء مجلد جديد باستخدام الأمر `mkdir` كما يلي:

```
mkdir environments
cd environments
```

بعد أن انتقلتَ إلى المجلد الذي تريد احتواء البيئات فيه، تستطيع الآن إنشاء بيئة جديدة بتنفيذ الأمر الآتي:

```
python3 -m venv my_env
```

سنستعمل الاسم my_env مجازًا للبيئة ولكن يجب أن تختار اسمًا ذا معنًى يناسب المشروع.

سيُنشِئ الأمر pyvenv مجلدًا جديدًا فيه بعض الملفات التي يمكن عرضها باستخدام الأمر ls:

```
ls my_env
```

ستظهر لك مخرجات شبيهة بالمخرجات التالية:

```
bin include lib lib64 pyvenv.cfg
```

تعمل هذه الملفات مع بعضها لضمان أنَ تكون مشاريعك معزولةٌ عن سياق الآلة المحلية، لكي لا تختلط ملفات النظام مع ملفات المشاريع. وهذا أمرٌ حسنٌ لإدارة الإصدارات ولضمان أنَّ كل مشروع يملك وصولًا إلى الحزمِ التي يحتاج إليها.

عليك تفعيل البيئة لاستخدامها، وذلك بكتابة الأمر التالي الذي سيُنفِّذ سكربت التفعيل:

```
source my_env/bin/activate
```

يجب أن تظهر الآن سابقةٌ (prefix) في المِحث (prompt) والتي هي اسم البيئة المستخدمة، وفي حالتنا هذه يكون اسمها my_env:

```
(my_env) [sammy@centosserver environments]$
```

ستسمح لك السابقة بمعرفة أنَّ البيئة my_env مفعلة حاليًا، وهذا يعني أنَّنا سنستخدم إعدادات وحزم هذه البيئة عند إنشاء مشاريع جديدة.

لاحظ أن مدير حزم بايثون pip قد ثُبِّت مسبقًا والذي سنستعمله لتثبيت الحزم البرمجية

وإدارتها في بيئتنا البرمجية السابقة، فيمكننا تثبيت أي حزمة بتنفيذ الأمر التالي:

```
(my_env) [sammy@centosserver environments]$ sudo pip install
package_name
```

يشير `package_name` هنا إلى أي حزمة أو مكتبة من بايثون مثل Django الحزمة المخصصة لتطوير الويب أو NumPy الحزمة المخصصة لإجراء الحسابات الرياضية المتقدمة، فإذا أردت تثبيت الحزمة الأخيرة، يمكنك ببساطة تنفيذ الأمر `pip install numpy`.

يجب أن تكون بيئتك الافتراضية جاهزةً للاستخدام بعد اتباعك للخطوات السابقة.

ملاحظة: يمكنك داخل البيئة الافتراضية أن تستخدم الأمر `python` بدلًا من `python3.6` والأمر `pip` بدلًا من `pip3.6` إن شئتَ أما إذا كنتَ تستخدم بايثون3 خارج البيئة الافتراضية، فيجب عليك حينها استخدام `python3.6` و `pip3.6` حصرًا.

---

٥. إنشاء برنامج بسيط

بعد أن أكملنا ضبط بيئتنا الافتراضية، لننشِئ برنامجًا بسيطًا يعرض العبارة «مرحبا بالعالم!»، وبهذا سنتحقّق من أنّ البيئة تعمل بالشكل الصحيح، ولكي تتعوّد على إنشاء برامج بايثون إن كنتَ وافدًا جديدًا على اللغة.

علينا أولًا تشغيل محرر ملفات نصية لإنشاء ملف جديد، وليكن المحرر vi:

```
(my_env) [sammy@centosserver environments]$ vi hello.py
```

بعد فتح الملف في نافذة الطرفية، اكتب الحرف `i` للدخول إلى وضع الإدراج (insert mode)، بعدها يمكننا كتابة البرنامج:

```
print("Hello, World!")
```

الآن اضغط على الزر ESC للخروج من وضع الإدراج. بعد ذلك ، اكتب x: ثم ENTER لحفظ الملف وإغلاقه.

نحن جاهزون الآن لتنفيذ البرنامج:

```
(my_env) [sammy@centosserver environments]$ python hello.py
```

سيؤدي برنامج hello.py الذي أنشأتَه إلى طباعة الناتج التالي في الطرفية:

```
Hello, World!
```

للخروج من البيئة، اكتب الأمر deactivate وستعود إلى مجلدك الأصلي.

تهانينا! لقد ضبطتَ الآن بيئة تطويرٍ للغة بايثون 3 في CentOS 8، حان الآن الوقت للتعمق بلغة بايثون وإنشاء برامج رائعة! انتقل إلى الفصل التالي، استخدام سطر أوامر بايثون التفاعلي.

5. macOS

سيُرشِدُك هذا القسم خطوةً بخطوة إلى كيفية تثبيت بايثون 3 في macOS، وتثبيت بيئة برمجة عبر سطر الأوامر.

ا. المتطلبات المسبقة

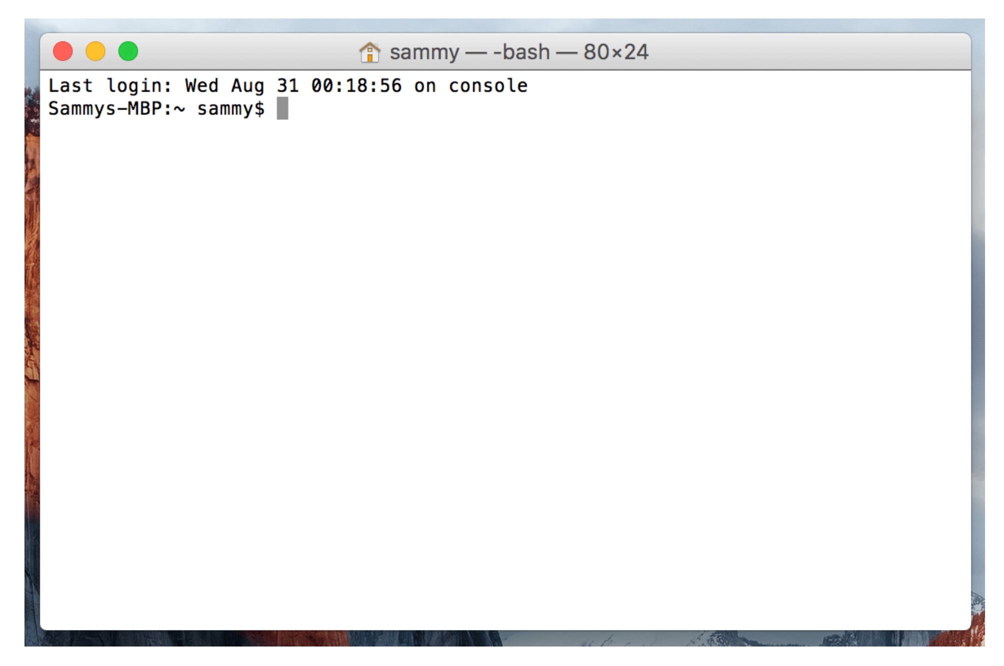
يجب أن تملك جهازًا عليه نظام macOS متصل بالشبكة مع صلاحيات مدير (administrative access).

إذا لم تكن لك خبرة في التعامل مع بيئة النافذة الطرفية، فيمكنك مطالعة المقال «مدخل إلى طرفيّة لينكس Linux Terminal».

ب. فتح نافذة الطرفية

سنجري معظم أطوار التثبيت والإعدادات عبر سطر الأوامر، والذي هو طريقةٌ غيرُ رسوميةٍ للتعامل مع الحاسوب، فبدلًا من الضغط على الأزرار، ستكتب نصًّا وتعطيه للحاسوب لينفذه، وسيُظهِر لك ناتجًا نصيًا أيضًا. يمكن أن يساعدك سطر الأوامر على تعديل أو أتمتة مختلف المهام التي تنجزها على الحاسوب يوميًا، وهو أداةٌ أساسيةٌ لمطوري البرمجيات.

طرفية ماك (macOS Terminal) هي تطبيق يمكنك استخدامه للدخول إلى واجهة سطر الأوامر. مثل التطبيقات الأخرى، ستجد تطبيق الطرفية بالذهاب إلى Finder، وفتح المجلد Applications، ثم الذهاب إلى المجلد Utilities، ثم النقر المزدوج على Terminal لفتحه. أو يمكنك استخدام Spotlight عبر الضغط على الزرّين `command` و `spacebar` للعثور على Terminal بكتابته في المربع الذي سيظهر.



هنالك الكثير من الأوامر التي عليك تعلمها لكي تتمكن من الاستفادة منه. هنالك مقالات في أكاديمية حسوب (مثل مقال مدخل إلى طرفيّة لينكس Linux Terminal) ستعلمك أساسيات سطر الأوامر، وهنالك كتاب «سطر أوامر لينكس» الذي يُعتَبر مرجعا لطريقة التعامل مع سطر الأوامر في لينكس، والذي يشبه نظيره في ماك.

ج. تثبيت Xcode

Xcode هي بيئة تطوير متكاملة (IDE) تتألف من أدوات تطوير البرامج لنظام التشغيل MacOS. قد يكون Xcode مثبتًا عندك سلفًا. للتحقق من ذلك، اكتب في نافذة الطرفية ما يلي:

```
xcode-select -p
```

إن حصلت على المخرجات التالية، فهذا يعني أنّ مثبت Xcode:

```
/Library/Developer/CommandLineTools
```

إن تلقيت خطأ، فثبّت Xcode من المتجر App Store واعتمد الخيارات الافتراضيَّة.

بعد تثبيت Xcode، ارجع إلى النافذة الطرفيَّة. ثم ثبّت التطبيق Command Line Tools عن طريق كتابة:

```
xcode-select --install
```

عند هذه المرحلة، يكون قد ثبّت Xcode، والتطبيق Command Line Tools الخاص به، ونحن الآن مستعدون لتثبيت مدير الحزم Homebrew.

د. تثبيت وإعداد Homebrew

في حين أنَّ طرفية macOS تتمتع بالكثير من وظائف طرفية لينكس وأنظمة يونكس الأخرى، إلا أنها لا تأتي بمدير حزم جيد. مدير الحزم (package manager) هو مجموعة من أدوات البرمجيات التي تعمل على أتمتة عمليات التثبيت، بما في ذلك التثبيت الأولي للبرامج، وترقيتها، وإعدادها، وإزالتها عند الحاجة. تحفظ هذه الأدوات التثبيتات في موقع مركزي، ويمكنها صيانة جميع حزم البرامج على النظام وفق تنسيقات (formats) معروفة. توفر Homebrew لنظام التشغيل macOS نظام إدارة حزم مجاني ومفتوح المصدر يبسط عملية تثبيت البرنامج. لتثبيت Homebrew، نفّذ في الطرفية الأمر التالي:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/insta
ll)"
```

طُوِّر Homebrew عبر روبي، لذلك سيعدِّل مسار روبي على جهازك. يسحب الأمر `curl` السكربت من عنوان URL المحدد. سيوضح ذلك السكربت ما سيفعله، ثم يوقف العملية ليطلب منك التأكيد. يوفر لك هذا الكثير من الملاحظات حول ما سيفعله السكربت في نظامك، ويمنحك الفرصة للتحقق من العملية.

لاحظ أنَّك عند إدخال كلمة المرور، فلن تعرض الطرفية المحارف التي تكتبها، ولكنَّها ستُسجِّل، بعد إدخال كلمة المرور اضغط على الزر `return`. واضغط على الحرف `y` إن طُلِب منك تأكيد التثبيت.

لنلق نظرة على الرايات (flags) المرتبطة بالأمر `curl`:

- تخبر الرايتان `-f` أو `--fail` الطرفية بعدم إعطاء مخرجات على هيئة مستند HTML عند حدوث أخطاء في الخادم.
- تُصمِت الرايتان `-s` أو `--silent` الأمر `curl`، بمعنى أنه لن يعرض معلومات التقدم، وعند جمعها مع الرايتين `-S` أو `--show-error`، ستجعل `curl` تُظهر رسالة خطأ في حال الفشل.
- تطلب الرايتان `-L` أو `--location` من `curl` إعادة الطلبية (request) إلى مكان جديد إذا أبلغ الخادم بأنَّ الصفحة المطلوبة قد نُقِلت إلى موقع أخر.

بمجرد اكتمال عملية التثبيت، سنضع مجلد Homebrew في أعلى متغير البيئة PATH. سيضمن ذلك أن يتم استدعاء عمليات تثبيت Homebrew عبر الأدوات التي قد يختارها نظام التشغيل Mac OS X تلقائيًا، والتي قد تتعارض مع بيئة التطوير التي تنشئها.

يجب عليك إنشاء أو فتح الملف ‎~/.bash_profile‎ باستخدام محرر نصوص سطر الأوامر nano باستخدام الأمر nano:

```
nano ~/.bash_profile
```

بعد فتح الملف في نافذة الطرفية، اكتب ما يلي:

```
export PATH=/usr/local/bin:$PATH
```

لحفظ التغييرات، اضغط على المفتاح control والحرف o بالتزامن، وعند مطالبتك بالتأكيد، اضغط على المفتاح return. يمكنك الآن الخروج من nano عن طريق الضغط على المفتاح control والحرف x بالتزامن.

لتفعيل هذه التغييرات، اكتب في نافذة الطرفية:

```
source ~/.bash_profile
```

ستصير الآن التغييرات التي أجريتها على متغير البيئة PATH فعالة. يمكننا التحقق من أنّ Homebrew قد ثُبِّت بنجاح عن طريق كتابة:

```
brew doctor
```

إذا لم تكن هناك حاجة إلى أي تحديثات في هذا الوقت، فستطبع الطرفية ما يلي:

```
Your system is ready to brew.
```

خلاف ذلك، قد تحصل على تنبيه يدعوك إلى تنفيذ أمر آخر مثل brew update للتأكد من أنّ Homebrew محدَّث. بمجرد أن تصبح Homebrew جاهزة، يمكنك تثبيت بايثون 3.

٥. تثبيت بايثون 3

يمكنـــك اســـتخدام Homebrew للبحث عن الـــبرامج الـــتي يمكنـــك تثبيتهـــا عـــبر الأمر brew search، ويمكنك الحصول على قائمة الحزم والوحدات ذات العلاقة ببايثون فقط عبر تنفيذ الأمر التالي:

```
brew search python
```

نافذة الطرفية ستخرج قائمة من الحزم والوحدات التي يمكنك تثبيتها على النحو التالي:

```
app-engine-python          micropython                python3
boost-python               python                     wxpython
gst-python                 python-markdown            zpython
homebrew/apache/mod_python               homebrew/versions/gst-
python010
homebrew/python/python-dbus              Caskroom/cask/kk7ds-
python-runtime
homebrew/python/vpython                  Caskroom/cask/mysql-
connector-python
```

سيكون بايثون 3 من بين العناصر المدرجة في القائمة. لذلك دعنا نثبّتها:

```
brew install python3
```

ستعرض لك نافذة الطرفيّة ملاحظات بشأن عملية تثبيت بايثون 3، وقد يستغرق الأمر بضع دقائق قبل اكتمال التثبيت.

إلى جانب بايثون 3، ستثبّت Homebrew و pip و setuptools و wheel. سنستخدم pip، وهي أداةٌ تعمل مع بايثون تُثبِّت وتدير الحزم البرمجية التي قد نحتاج إلى استخدامها في تطوير مشاريعنا (سنتعلم المزيد عن الوحدات والحزم في الفصل اللاحق المخصّص للوحدات).

يمكن تثبيت حزم بايثون بكتابة ما يلي مع تبديل package_name باسم الوحدة:

```
pip3 install package_name
```

عليك وضع اسم الحزمة أو المكتبة التابعة لبايثون مكان `package_name` مثل Django لتطوير الويب، أو NumPy لإجراء الحسابات العلمية. لذا، إن شئتَ تنزيل NumPy فيمكنك تنفيذ الأمر `pip3 install numpy`.

تسهّل الأداة setuptools تحزيم مشاريع بايثون، أما wheel فهي تنسيق حزم (built-package format) لبايثون يمكنه تسريع إنتاجية البرامج عن طريق تقليل عدد المرات التي تحتاج فيها إلى التصريف.

بعد إكمال العملية، يمكننا التحقق من إصدار بايثون 3 المُثبَّت في النظام بكتابة:

```
python3 --version
```

ستحصل على مخرجات في نافذة الطرفية والتي ستريك إصدار بايثون المثبَّت. والذي سيكون افتراضيًا أحدث إصدار مستقر ومتاح من بايثون 3.

لتحديث إصدار بايثون 3، يمكنك أولاً تحديث Homebrew، ثمَّ تحديث بايثون:

```
brew update
brew upgrade python3
```

من الممارسات الجيدة تحديث إصدار بايثون الذي تعمل به من حين لآخر.

و. إعداد بيئة افتراضية

الآن بعد تثبيت Xcode و Homebrew وبايثون، يمكننا المضي قدمًا لإنشاء بيئة البرمجة خاصتنا.

تُمكّنك البيئات الافتراضية من إنشاء مساحة معزولة في حاسوبك مخصصة لمشاريع بايثون، مما يعني أنَّ كل مشروع تعمل عليه ستكون له اعتماديّاته (dependencies) الخاصة به، والتي لن تؤثِّر على غيره من المشاريع.

يوفِّر لنا ضبط بيئةٍ برمجيةٍ تحكمًا أكبر بمشاريع بايثون، وإمكانية التعامل مع إصداراتٍ مختلفةٍ من حزم بايثون. وهذا مهمٌ كثيرًا عندما تتعامل مع الحزم الخارجية.

يمكنك ضبط أيِّ عددٍ تشاء من البيئات الافتراضية، وكل بيئة ستكون ممثلة بمجلد في حاسوبك يحتوي على عدد من السكربتات.

اختر المجلد الذي تريد أن تضع فيه بيئات بايثون، أو يمكنك إنشاء مجلد جديد باستخدام الأمر `mkdir` كما يلي:

```
mkdir environments
cd environments
```

بعد أن انتقلتَ إلى المجلد الذي تريد احتواء البيئات فيه، تستطيع الآن إنشاء بيئة جديدة بتنفيذ الأمر التالي:

```
python3.6 -m venv my_env
```

سيُنشِئ هذا الأمر مجلدًا جديدًا (في هذه الحالة يُسمى `my_env`) فيه بعض الملفات:

- يشير الملف `pyvenv.cfg` إلى توزيعة بايثون التي استخدمتها لتنفيذ الأمر.
- يحتوي المجلد الفرعي `lib` نسخة من إصدار بايثون، ويحتوي على مجلد يسمى `site-packages`، والذي سيكون فارغًا في البداية، ولكنه سيُستخدم لتخزين وحدات الطرف الثالث التي ستثبِّتها.
- المجلد الفرعي `include` يُصرِّف (packages) الحزم.

- يحتوي المجلد الفرعي `bin` نسخة من رُقامة بايثون (Python binary) جنبا إلى جنب مع سكربت التفعيل (activate shell script) الذي سيُستخدم.

تعمل هذه الملفات مع بعضها لضمان أنَ تكون مشاريعك معزولةٌ عن سياق الآلة المحلية، لكي لا تختلط ملفات النظام مع ملفات المشاريع. وهذا أمرٌ حسنٌ لإدارة الإصدارات ولضمان أنَّ كل مشروع يملك وصولًا إلى الحزمِ التي يحتاج إليها.

عليك تفعيل البيئة لاستخدامها، وذلك بكتابة الأمر التالي الذي سيُنفِّذ سكربت التفعيل:

```
source my_env/bin/activate
```

يجب أن تظهر الآن سابقةٌ (prefix) في المِحث (prompt) والتي هي اسم البيئة المستخدمة، وفي حالتنا هذه يكون اسمها `my_env`، وهذا يعني أننا لن سنستخدم إلا إعدادات وحزم هذه البيئة عند إنشاء مشاريع جديدة.

يجب أن تكون بيئتك الافتراضية جاهزةً للاستخدام بعد اتباعك للخطوات السابقة.

ملاحظة: يمكنك داخل البيئة الافتراضية أن تستخدم الأمر `python` بدلًا من `python3` والأمر `pip` بدلًا من `pip3` إن شئتَ أما إذا كنتَ تستخدم بايثون3 خارج البيئة الافتراضية، فيجب عليك حينها استخدام `python3` و `pip3` حصرًا. لأنَّ `python` و`pip` ستستدعيان إصدارًا قديمًا من بايثون

ز. إنشاء برنامج بسيط

بعد أن أكملنا ضبط بيئتنا الافتراضية، لننشِئ برنامجًا بسيطًا يعرض العبارة «مرحبا بالعالم!»، وبهذا سنتحقق من أنَّ البيئة تعمل بالشكل الصحيح، ولكي تتعوَّد على إنشاء برامج بايثون إن كنتَ وافدًا جديدًا على اللغة.

علينا أولًا تشغيل محرر ملفات نصية لإنشاء ملف جديد، وليكن المحرر nano الذي يعمل من سطر الأوامر:

```
(my_env) sammys-MBP:~ sammy$ nano hello.py
```

بعد فتح الملف في نافذة الطرفية، سنكتب البرنامج الخاص بنا:

```
print("Hello, World!")
```

أغلق محرر nano بالضغط على Ctrl+x ثم اضغط على y عندما يسألك عن حفظ الملف.

بعد أن يُغلَق المحرر nano وتعود إلى سطر الأوامر، حاول تنفيذ البرنامج:

```
(my_env) sammys-MBP:~ sammy$  python hello.py
```

سيؤدي برنامج `hello.py` الذي أنشأتَه إلى طباعة الناتج التالي في الطرفية:

```
Hello, World!
```

للخروج من البيئة، اكتب الأمر `deactivate` وستعود إلى مجلدك الأصلي.

تهانينا! لقد ضبطتَ الآن بيئة تطويرٍ للغة بايثون 3 في نظام Mac OS X، حان الآن الوقت للتعمق بلغة بايثون وإنشاء برامج رائعة!

3

سطر أوامر بايثون التفاعلي

يوفر سطر أوامر بايثون التفاعلي (Python interactive console)، ويسمى أيضًا مترجم بايثون (Python interpreter) للمبرمجين طريقة سريعة لتنفيذ الأوامر، وتجربة أو اختبار التعليمات البرمجية دون الحاجة إلى إنشاء وكتابة أي شيفرة برمجية.

يمكن الوصول من خلال سطر الأوامر التفاعلي إلى جميع دوال بايثون المُضمّنة، وجميع الوحدات المُثبَّتة، وتاريخ الأوامر، والإكمال التلقائي. ويوفر سطر الأوامر التفاعلي الفرصة لاستكشاف وتجربة شيفرات تعليمات بايثون، والقدرة على نسخ الشيفرة البرمجية ولصقها في ملف الشيفرة المصدرية عندما تصبح جاهزة أي بعد تجريبها والتأكد من عملها.

سيتناول هذا الفصل كيفية العمل بسطر الأوامر التفاعلي لبايثون، وكيفية الاستفادة منه أثناء كتابة الشيفرة.

1. فتح سطر الأوامر التفاعلي

يمكن الوصول إلى سطر الأوامر التفاعلي من أيّ حاسوب محلي أو خادم مثبَّت فيه بايثون. التعليمة التي ستستخدمها عمومًا للدخول إلى سطر الأوامر التفاعلي في الإصدار الافتراضي لبايثون هي:

```
python
```

إذا أعددّت البيئة البرمجية وجهزتها وفق إرشادات الفصل السابق، فيمكنك إنشاء بيئة واستعمال إصدار بايثون والوحدات المُثبَّتة فيها عن طريق الدخول أولًا إلى تلك البيئة (إن لم تُهيِّئ البيئة الوهمية بعد، فعد إلى الفصل السابق وجهِّز البيئة الوهمية قبل تنفيذ الأوامر التالية):

```
cd environments
. my_env/bin/activate
```

ثم اكتب الأمر python:

```
(my_env) sammy@ubuntu:~/environments$ python
```

في مثالنا الحالي، الإصدار الافتراضي هو Python 3.7.7، والذي يُعرَض في المخرجات بمجرد إدخال الأمر، إلى جانب إشعار حقوق الملكية، وبعض الأوامر التي يمكنك كتابتها للحصول على معلومات إضافية:

```
Python 3.7.7 (default, Jun  4 2020, 15:43:14)
[GCC 9.3.1 20200408 (Red Hat 9.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

في بداية كل سطر ستجد ثلاث علامات "أكبر من" (>>>):

```
>>>
```

يمكنك استهداف إصدارات محددة من بايثون عن طريق إلحاق رقم الإصدار بالأمر، وبدون مسافات مثل تنفيذ الأمر python2.7:

```
Python 2.7.18 (default, Apr 20 2020, 00:00:00)
[GCC 9.3.1 20200408 (Red Hat 9.3.1-2)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

تبيِّن المخرجات لنا أنَّنا نستخدم الإصدار Python 2.7.17. إذا كان هذا هو الإصدار الافتراضي لبايثون 2، فيمكننا أيضًا الدخول إلى سطر الأوامر التفاعلي باستخدام الأمر python2.

بالمقابل، يمكننا استدعاء إصدار بايثون 3 الافتراضي باستخدام الأمر python3 فنحصل على المخرجات التالية:

```
Python 3.7.7 (default, Jun  4 2020, 15:43:14)
[GCC 9.3.1 20200408 (Red Hat 9.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

يمكن أيضًا أن نفتح سطر الأوامر التفاعلي أعلاه باستخدام الأمر python3.7. بعد تشغيل سطر الأوامر التفاعلي لبايثون، يمكننا المضي قدمًا والبدء في العمل.

2. العمل في سطر أوامر بايثون التفاعلي

يقبل مترجم بايثون التفاعلي (Python interactive interpreter) قواعد لغة بايثون، والتي تضعها بعد البادئة <<<.

يمكننا، على سبيل المثال، إنشاء متغيّر وإسناد قيمة له بالشكل التالي:

```
>>> birth_year = 1868
```

بمجرد تعيين قيمة العدد الصحيح 1868 إلى المتغيّر birth_year، سنضغط على زر الإدخال ونحصل على سطر جديد يبدأ بثلاث علامات "أكبر من" (<<<):

```
>>> birth_year = 1868
>>>
```

يمكننا الاستمرار في تعيين المتغيّرات، وإجراء الحسابات الرياضياتية:

```
>>> birth_year = 1868
>>> death_year = 1921
>>> age_at_death = death_year - birth_year
>>> print(age_at_death)
53
>>>
```

كما نفعل في ملفات البرامج النصية (السكربتات)، أنشأنا متغيرات جديدة أخرى وأسندنا قيمةً لها تناسب اسمها، ثم طرحنا قيمة متغيِّرٍ من آخر، وطلبنا من سطر الأوامر طباعة المتغير الذي يمثِّل الفرق عبر الدالة `print()`.

يمكنك أيضًا استخدام سطر الأوامر التفاعلي كآلة حاسبة:

```
>>> 203 / 20
10.15
>>>
```

هنا، قسمنا العدد الصحيح 203 على 20، لنحصل على الناتج 10.15.

3. تعدُّد الأسطر

عندما نكتب شيفرة متعددة الأسطر، سيستخدم المترجم أسطر الاستمرارية، وهي أسطر مسبوقة بثلاث نقاط `...` وللخروج من هذا الوضع، ستحتاج إلى الضغط على الزر ENTER مرتين.

تعيِّن الشيفرة التالية قيمتي متغيِّرين، ثم تستخدم تعليمة شرطيَّة لتحديد ما يجب طباعته:

```
>>> sammy = 'Sammy'
>>> shark = 'Shark'
>>> if len(sammy) > len(shark):
...     print('Sammy codes in Java.')
... else:
...     print('Sammy codes in Python.')
...
Sammy codes in Python.
>>>
```

في هذه الحالة، يتساوى طولا السلسلتين النصيتين، لذلك يتم تنفيذ التعليمة `else`.

لاحظ أنَّك ستحتاج إلى الحفاظ على مسافة بادئة بايثون (Python indenting) المؤلفة من أربعة مسافات بيضاء، وإلا سيُطلَق خطأٌ:

```
>>> if len(sammy) > len(shark):
... print('Sammy codes in Java.')
  File "<stdin>", line 2
    print('Sammy codes in Java.')
        ^
IndentationError: expected an indented block
>>>
```

إضافة إلى تجربة التعليمات البرمجية متعددة الأسطر في سطر الأوامر، يمكنك أيضًا استيراد الوحدات.

4. استيراد الوحدات

يوفر لك مترجم بايثون طريقةً سريعةً للتحقق مما إذا كانت وحدات معيَّنة متوفِّرة في بيئة البرمجة الحاليَّة. يمكنك ذلك باستخدام التعليمة `import`:

```
>>> import matplotlib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'matplotlib'
```

في الحالة أعلاه، لم تكن الوحدة `matplotlib` متاحة في بيئة البرمجة الحالية. لتثبيت تلك الوحدة، ستحتاج إلى ترك المترجم التفاعلي، وتثبيتها باستخدام أداة إدارة الحزم pip مثل العادة:

```
(my_env) sammy@ubuntu:~/environments$ pip install matplotlib
Collecting matplotlib
  Downloading matplotlib-2.0.2-cp35-cp35m-manylinux1_x86_64.whl
(14.6MB)
...
Installing collected packages: pyparsing, cycler, python-
dateutil, numpy, pytz, matplotlib
Successfully installed cycler-0.10.0 matplotlib-2.0.2 numpy-
1.13.0 pyparsing-2.2.0 python-dateutil-2.6.0 pytz-2017.2
```

بمجرد تثبيت الوحدة matplotlib هي وجميع تبعياتها بنجاح، يمكنك العودة إلى المترجم التفاعلي:

```
(my_env) sammy@ubuntu:~/environments$ python
```

في هذه المرحلة، لن تتلقى أيَّ رسالة خطأ إن استوردت الوحدة، ويمكنك استخدام الوحدة المثبتة إما داخل سطر الأوامر، أو داخل ملف.

5. الخروج من سطر أوامر بايثون التفاعلي

هناك طريقتان رئيسيتان للخروج من سطر الأوامر التفاعلي: إما استخدام اختصار لوحة المفاتيح، أو استخدام دالة من دوال بايثون.

اختصار لوحة المفاتيح هو CTRL+D في أنظمة *نيكس (أي لينكس ويونكس)، أو CTRL+Z ثم CTRL في أنظمة ويندوز، وبذلك ستخرج من سطر الأوامر، وتعود إلى البيئة الطرفيَّة الأصليَّة:

```
...
>>> age_at_death = death_year - birth_year
>>> print(age_at_death)
53
>>>
sammy@ubuntu:~/environments$
```

بدلًا من ذلك، ستنهي الدالة quit() سطر الأوامر التفاعلي، وتعيدك إلى بيئة المحطة الطرفية الأصلية التي كنت فيها سابقًا:

```
>>> octopus = 'Ollie'
>>> quit()
sammy@PythonUbuntu:~/environments$
```

في حال استخدام الدالة quit()، فستُؤرَّخ في ملف التاريخ (history file)، بالمقابل لن يُسجَّل اختصار لوحة المفاتيح CTRL+D ذلك:

```
# /home/sammy/.python_history الملف
...
age_at_death = death_year - birth_year
print(age_at_death)
octopus = 'Ollie'
quit()
```

يمكن إنهاء مترجم بايثون بكلا الطريقتين، فاختر ما يناسبك.

6. الاطلاع على التاريخ

من فوائد سطر الأوامر التفاعلي أنّ جميع أوامرك تؤرَّخ في الملف .python_history في أنظمة *ينكس، بحيث يمكنك الاطلاع عليها في أيّ محرر نصي، مثل nano:

```
nano ~/.python_history
```

بمجرد فتحه باستخدام محرر نصوص، سيبدو ملف تأريخ بايثون الخاص بك على هذا النحو:

```
# /home/sammy/.python_history الملف
import pygame
quit()
if 10 > 5:
   print("hello, world")
else:
   print("nope")
sammy = 'Sammy'
shark = 'Shark'
...
```

بمجرد الانتهاء من ملفك، يمكنك الضغط على CTRL+X للخروج.

من خلال تتبع الأحداث المُؤرَّخة في بايثون، يمكنك الرجوع إلى الأوامر والتجارب السابقة، ونسخ ولصق أو تعديل الشيفرة لاستخدامها في الملفات البرمجية أو في Jupyter Notebook.

7. خلاصة الفصل

سطر الأوامر التفاعلي هو فضاءٌ لتجربة شيفرة بايثون، إذ يمكنك استخدامه أداة للاختبار والتجريب وغير ذلك.

لتنقيح (Debug) ملفات البرمجة في بايثون، يمكنك استخدام الوحدة code لفتح مترجم تفاعلي داخل ملف، وسنتحدث عن ذلك بالتفصيل في الفصل اللاحق: كيفية تنقيح بايثون باستخدام سطر الأوامر التفاعلي.

4

التعليقات واستخداماتها

التعليقات هي عبارات دخيلة على الشيفرات البرمجية وليست جزءًا منها، إذ تتجاهلها المُصرِّفات (compilers) والمترجمات (interpreters). يُسهِّل تضمين التعليقات في الشيفرات من قراءتها وفهمها ومعرفة وظيفة كل جزء من أجزائها، لأنَّها توفر معلومات وشروحات حول ما يفعله كل جزء من البرنامج.

بناءً على الغرض من البرنامج، يمكن أن تكون التعليقات بمثابة مُذكَّرات لك، أو يمكنك كتابتها لمساعدة المبرمجين الآخرين على فهم الشيفرة.

يُستحسَن كتابة التعليقات أثناء كتابة البرامج أو تحديثها، لأنَّك قد تنسى السياق وتسلسل الأفكار لاحقًا، والتعليقات القديمة قد تصبح عديمة الفائدة ومضللة إن لم تُحدَّث.

1. صياغة التعليقات

تبدأ التعليقات السطرية في بايثون بالعلامة # ومسافة بيضاء، وتستمر حتى نهاية السطر.

بشكل عام، ستبدو التعليقات على النحو التالي:

```
# هذا تعليق
```

نظرًا لأنَّ التعليقات لا تُنفَّذ، فعند تشغيل البرنامج، لن ترى أي إشارة للتعليقات، فالتعليقات توضع في الشيفرة المصدرية ليقرأها الناس، وليس للتنفيذ. في برنامج "Hello, World!" قد يبدو التعليق كما يلي:

```
# طبع "Hello, World!" في سطر الأوامر
print("Hello, World!")
```

في الحلقة for، قد تبدو التعليقات كما يلي:

```
# تعريف المتغير sharks كمصفوفة من السلاسل النصية
sharks = ['hammerhead', 'great white', 'dogfish', 'frilled',
'bullhead', 'requiem']

# حلقة For تمر على المصفوفة sharks
for shark in sharks:
  print(shark)
```

يجب أن تُحاذى التعليقات على نفس المسافة البادئة (indent) للشيفرة التي تعلَّق عليها. بمعنى أنَّ التعليقات على دالة ليس لها مسافة بادئة ستكون هي أيضًا بدون مسافة بادئة، وسيكون لكل مستوى من المسافات البادئة التالية تعليقات تتوافق مع الشيفرات البرمجية التي تعلِّق عليها.

على سبيل المثال، الشيفرة التالية تعرّف الدالة again() التي تسأل المستخدم إن كان يريد استخدام الحاسبة مجدّدًا، مع بعض التعليقات:

```
...
# تعريف الدالة again() لسؤال المستخدم
# إن كان يريد استخدام الحاسبة مجددا

def again():

   # أخذ المدخلات من المستخدم
   calc_again = input('''
Do you want to calculate again?
Please type Y for YES or N for NO.
''')

   # إن أدخل المستخدم Y فستُنفّذ الدالة calculate()
```

```
if calc_again == 'Y':
  calculate()

# إن كتب المستخدم N فقل وداعا للمستخدم وأنه البرنامج

elif calc_again == 'N':
  print('See you later.')

# إن أدخل المستخدم حرفًا آخر، فأعد تنفيذ الدالة
else:
  again()
```

الهدف من التعليقات هو مساعدة المبرمج الأصلي، أو أيِّ شخص آخر يستخدم مشروعه أو يتعاون معه، على فهمه. وإذا تعذّر صيانة التعليقات وتحديثها تحديثًا صحيحًا، وبالتوازي مع الشيفرة، فإنَّ عدم تضمين التعليقات قد يكون أفضل من كتابة تعليق يتناقض مع الشيفرة.

يجب أن تجيب التعليقات عن سؤال "لماذا" بدلًا من "ماذا" أو "كيف". لأنَّه ما لم تكن الشيفرة صعبة ومعقَّدة، فإن النظر إلى الشيفرة عمومًا كافٍ لفهم ما الذي تفعله الشيفرة، أو كيف تفعله.

2. التعليقات الكتلية

يمكن استخدام التعليقات الكتلية (Block Comments) لتوضيح الشيفرات البرمجية المعقَّدة التي لا تتوقع أن يكون القارئ على دراية بها. تنطبق هذه التعليقات الطويلة على جزء من الشيفرة أو جميعها، كما توضع في نفس مستوى المسافة البادئة للشيفرة.

في التعليقات الكتلية، يبدأ كل سطر بالعلامة # تليها بمسافة بيضاء واحدة. إذا كنت بحاجة إلى استخدام أكثر من فقرة واحدة، فيجب فصلها بسطر يحتوي على علامة # واحدة.

فيما يلي مثال على كتلة تعليقات تشرح ما يحدث في الدالة `main()` المعرفة أدناه:

```
# parser سوف تحلل الدالة main الوسائط عبر المتغير
# الوسائط ستُحدّد بواسطة المستخدم على سطر الأوامر. هذا سيمرر
# الوسيط word الذي يريد المستخدم تحليله مع اسم الملف
# المراد استخدامه، وكذلك تقديم نص مساعد إذا لم يُدخل
# المستخدم الوسيط بشكل صحيح
def main():
  parser = argparse.ArgumentParser()
  parser.add_argument(
    "word",
    help="the word to be searched for in the text file."
  )
  parser.add_argument(
    "filename",
    help="the path to the text file to be searched through"
  )
...
```

تُستخدَم التعليقات الكتلية عادةً عندما تكون الشيفرة غير واضحة، وتتطلب شرحًا شاملا.

يجب أن تتجنب الإفراط في التعليق على الشيفرة، ويجب أن تثق في قدرة المبرمجين الآخرين على فهم الشيفرة، إلا إذا كنت تكتب لجمهور معين.

3. التعليقات السطرية

توضع التعليقات السطرية (Inline comments) على نفس السطر الذي توجد فيه التعليمة البرمجية. ومثل التعليقات الأخرى، فإنّها تبدأ بالعلامة # ومسافة بيضاء واحدة.

بشكل عام، تبدو التعليقات السطرية كما يلي:

```
[code]  # تعليق مضمن حول الشيفرة
```

لا ينبغي الإكثار من استخدام التعليقات السطرية، ولكن عند استخدامها في محلها يمكن أن تكون فعالة لشرح الأجزاء الصعبة من الشيفرة. وقد تكون مفيدة أيضًا إن ظننت أنَّك قد لا تتذكر سطرًا من الشيفرة في المستقبل، أو إذا كنت تتعاون مع شخص قد لا يكون على دراية بجميع جوانب الشيفرة.

على سبيل المثال، إذا لم يكن هناك توضيح مسبق، فقد لا تعلم أنت أو المتعاونون معك أنَّ الشيفرة التالية تنشئ عددًا عِقديًّا، لذلك قد ترغب في إضافة تعليق مضمَّن:

```
z = 2.5 + 3j  # إنشاء عدد عقدي
```

يمكن أيضًا استخدام التعليقات السطريَّة لشرح السبب وراء فعل شيء ما، أو بعض المعلومات الإضافية، كما في المثال التالي:

```
x = 8  # ابتداء x بقيمة عشوائية
```

يجب استخدام التعليقات السطرية عند الضرورة وحسب، كما ينبغي أن توفِّر إرشادات مفيدة للشخص الذي يقرأ البرنامج.

4. تعليق جزء من الشيفرة بدواعي الاختبار والتنقيح

بالإضافة إلى استخدام التعليقات وسيلةً لتوثيق الشيفرة، يمكن أيضًا استخدام العلامة # لتعليق جزء من الشيفرة وتعطيله أثناء اختبار أو تنقيح البرنامج الذي تعمل عليه. أي عندما تواجه أخطاء بعد إضافة أسطر جديدة إلى الشيفرة، فقد ترغب في تعليق بعضها لمعرفة موضع الخلل. يمكن أن يتيح لك استخدام العلامة # تجربة بدائل أخرى أثناء إعداد الشيفرة. على سبيل المثال، قد تفاضل بين استخدام الحلقة `while` أو حلقة `for` أثناء برمجة لعبة، ويمكنك تعليق إحداهما بينما تختبر أيّهما أفضل:

```
import random

number = random.randint(1, 25)

# number_of_guesses = 0

for i in range(5):
# while number_of_guesses < 5:
   print('Guess a number between 1 and 25:')
   guess = input()
   guess = int(guess)

   # number_of_guesses = number_of_guesses + 1

   if guess < number:
     print('Your guess is too low')

   if guess > number:
     print('Your guess is too high')

   if guess == number:
     break

if guess == number:
   print('You guessed the number!')



else:
   print('You did not guess the number. The number was ' +
str(number))
```

يـتيح لـك تعليـق الشـيفرة البرمجيـة تجربـة عـدَّة طرائـق ومقاربـات برمجيَّة، بالإضـافة إلى مساعدتك على العثور على مكمن الخطأ من خلال التعليق المنهجي لبعض أجزاء البرنامج.

5. خلاصة الفصل

سيساعدك استخدام التعليقات في برامج بايثون على جعل برامجك أكثر مقروئيـة، سواءً لك أو لغيرك. التعليقات المناسبة وذات الصلة والمفيدة ستسهِّل تعاون الآخرين معك في مشاريع البرمجة وتجعل شيفرتك أكثر قيمة.

5

المتغيرات واستخداماتها

المتغـيـرات (variables) هي مفهــوم بــرمجي مهم. إنَّهــا رمــوز تــدل على البيانــات الــتي تستخدمها في برنامجك، أي تعد حاويات للبيانات التي ستتعامل معها في برنامج.

سيغطي هذا الفصل بعض أساسيات المتغيرات، وكيفيـة استخدامها استخدامًا صحيحًا في برامج بايثون 3.

1. فهم المتغيرات

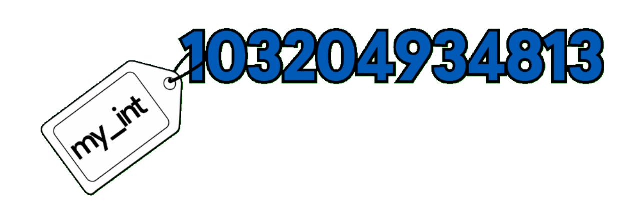
من الناحية الفنية، يُخصَّص للمتغير مساحة تخزينيـة في الذاكرة توضع القيمة المرتبطة به فيها. يُستخدم اسم المتغير للإشارة إلى تلك القيمة المُخزَّنة في ذاكرة البرنامج التي هي جزء من ذاكرة الحاسوب. المُتغيِّر أشبه بعنوان تُلصقه على قيمة مُعيَّنة:



لنفترض أنّ لدينا عددًا صحيحًا يساوي 103204934813، ونريد تخزينه في متغيِّر بدلًا من إعــادة كتابــة هــذا العــدد الطويــل كــل مــرة، لــذلك سنســتخدم شــيئًا يُســهِّل تــذكُّره، مثل المتغير my_int:

```
my_int = 103204934813
```

إذا نظرنا إليه على أنَّه عنوانٌ مرتبط بقيمة، فسيبدو على النحو التالي:



عنوان القيمة هو `my_int` المكتوب عليها، والقيمة هي 103204934813 (نوعها عدد صحيح، سنتطرق إلى أنواع البيانات في الفصل التالي).

التعليمة `my_int = 103204934813` هي تعليمة إسناد (assignment statement)، وتتألف من الأجزاء التالية:

- اسم المتغير (`my_int`)
- معامل الإسناد، المعروف أيضًا باسم "علامة المساواة" (`=`)
- القيمة التي أُسنِدَت إلى اسم المتغير (103204934813)

تشكل هذه الأجزاء الثلاثة معًا التعليمة التي تُسنِد على المتغير `my_int` القيمة العددية الصحيحة 103204934813 .

بمجرد تعيين قيمة متغيِّر ما، نكون قد هيَّئنا أو أنشأنا ذلك المتغير. وبعد ذلك، يمكننا استخدام ذلك المتغير بدلًا من القيمة. في بايثون، لا يلزم التصريح عن المتغيِّرات قبل استخدامها كما هو الحال في بعض لغات البرمجة الأخرى، إذ يمكنك البدء في استخدام المتغير مباشرةً.

بمجرد إسناد القيمة 103204934813 إلى المتغير `my_int`، يمكننا استخدام `my_int` مكان العدد الصحيح، كما في المثال التالي:

```
print(my_int)
```

والمخرجات هي:

```
103204934813
```

استخدام المتغيرات يسهل علينا إجراء العمليات الحسابية. في المثال التالي، سنستخدم التعليمة السابقة، `my_int = 1040` ، ثم سنطرح من المتغير `my_int` القيمة 813 :

```
print(my_int - 813)
```

وسينتج لنا:

```
103204934000
```

في هذا المثال، يجري بايثون العملية الحسابية، ويطرح 813 من المتغير `my_int` ، ويعيد القيمة 103204934000.

يمكن ضبط المتغيرات وجعلها تساوي ناتج عملية حسابية. دعنا نجمع عددين معًا، ونخزِّن قيمة المجموع في المتغير x:

```
x = 76 + 145
```

يشبه المثال أعلاه إحدى المعادلات التي تراها في كتب الجبر. في الجبر، تُستخدَم الحروف والرموز لتمثيل الأعداد والكميات داخل الصيغ والمعادلات، وبشكل مماثل، المتغيرات أسماء رمزية تمثل قيمة من نوع بيانات معيَّن. الفرق في لغة بايثون،أنّ َعليك التأكد دائمًا من وضع المتغير على الجانب الأيسر من المعادلة.

لنطبع x:

```
print(x)
```

والمخرجات:

```
221
```

أعادت بايثون القيمة 221 لأنَّه أُسنِد إلى المتغير x مجموع 76 و 145.

يمكن أن تحوي المتغيِّرات أي نوع من أنواع البيانات، وليس الأعداد الصحيحة فقط:

```
my_string = 'Hello, World!'
my_flt = 45.06
my_bool = 5 > 9 # True أو False القيم المنطقية ستعيد إما
my_list = ['item_1', 'item_2', 'item_3', 'item_4']
my_tuple = ('one', 'two', 'three')
my_dict = {'letter': 'g', 'number': 'seven', 'symbol': '&'}
```

إذا طبعت أيًّا من المتغيِّرات المذكورة أعلاه، فستعيد بايثون قيمة المتغيِّر. على سبيل المثال، في الشيفرة التالية سنطبع متغيرًا يحتوي قائمة:

```
my_list = ['item_1', 'item_2', 'item_3', 'item_4']
print(my_list)
```

وسينتج لنا:

```
['item_1', 'item_2', 'item_3', 'item_4']
```

لقد مرَّرنا القيمة ['item_1' و 'item_2' و 'item_3' و 'item_4'] إلى المتغير my_list، ثم استخدمنا الدالة print() لطباعة تلك القيمة باستدعاء my_list.

تُخصِّص المتغيِّرات مساحةً صغيرةً من ذاكرة الحاسوب، وتقبل قيمًا تُوضَع بعد ذلك في تلك المساحة.

2. قواعد تسمية المتغيرات

تتسم تسمية المتغيرات بمرونة عالية، ولكن هناك بعض القواعد التي عليك أخذها في الحسبان:

- يجب أن تكون أسماء المتغيرات من كلمة واحدة فقط (بدون مسافات)
- يجب أن تتكوّن أسماء المتغيرات من الأحرف والأرقام والشرطة السفلية (_) فقط
- لا ينبغي أن تبدأ أسماء المتغيرات برقم

باتباع القواعد المذكورة أعلاه، دعنا نلقي نظرة على بعض الأمثلة:

اسم غير صحيح	اسم صحيح	لماذا غير صالح؟
my-int	my_int	غير مسموح بالشرطات
4int	int4	لا يمكن البدء برقم
$MY_INT	MY_INT	لا يمكن استخدام أيّ رمز غير الشرطة السفلية
another int	another_int	لا ينبغي أن يتكون الاسم من أكثر من كلمة واحدة

من الأمور التي يجب أخذها في الحسبان عند تسمية المتغيرات، هو أنَّها حساسة لحالة الأحرف، وهذا يعني أنَّ my_int و MY_INT و My_Int و mY_iNt كلها مختلفة. ينبغي أن تتجنب

استخدام أسماء متغيِّرات متماثلة لضمان ألا يحدث خلط عندك أو عند المتعاونين معك، سواء الحاليين والمستقبليِّين.

أخيرًا، هذه بعض الملاحظات حول أسلوب التسمية. من المتعارف عليه عند تسمية المتغيرات أن تبدأ اسم المتغير بحرف صغير، وأن تستخدم الشرطة السفلية عند فصل الكلمات. البدء بحرف كبير مقبول أيضًا، وقد يفضِّل بعض الأشخاص استعمال تنسيق سنام الجمل (camelCase، الخلط بين الأحرف الكبيرة والصغيرة) عند كتابة المتغيرات، ولكنَّ هذه الخيارات أقل شهرة.

تنسيق غير شائع	تنسيق شائع	لماذا غير متعارف عليه
myInt	my_int	أسلوب سنام الجمل (camelCase) غير شائع
Int4	int4	الحرف الأول كبير
myFirstString	my_first_string	أسلوب سنام الجمل (camelCase) غير شائع

الخيار الأهم الذي عليك التمسك به هو الاتساق. إذا بدأت العمل في مشروع يستخدم تنسيق سنام الجمل في تسمية المتغيرات، فمن الأفضل الاستمرار في استخدام ذلك التنسيق.

يمكنك مراجعة توثيق تنسيق الشيفرات البرمجية في بايثون في موسوعة حسوب للمزيد من التفاصيل.

3. تغيير قيم المتغيرات

كما تشير إلى ذلك كلمة "متغيِّر"، يمكن تغيير قيم المتغيرات في بايثون بسهولة. هذا يعني أنَّه يمكنك تعيين قيمة مختلفة إلى متغير أُسنِدَت له قيمة مسبقًا بسهولة بالغة.

القدرة على إعادة إسناد القيم مفيدة للغاية، إذ قد تحتاج خلال أطوار برنامجك إلى قبول قيم ينشئها المستخدم وتُحيلها على متغير.

سهولة إعادة إسناد المتغيرات مفيدة أيضًا في البرامج الكبيرة التي قد تحتاج خلالها إلى تغيير القيم باستمرار.

سنُسند إلى المتغير x أولًا عددًا صحيحًا، ثم نعيد إسناد سلسلة نصية إليه:

```
# تعيين x إلى قيمة عددية
x = 76
print(x)

# إعادة تعيين x إلى سلسلة نصية
x = "Sammy"
print(x)
```

وسينتج لنا:

```
76
Sammy
```

يوضح المثال أعلاه أنه يمكننا أولًا إسناد قيمة عددية إلى المتغير x، ثم إعادة إسناد قيمة نصية إليه.

إذا أعدنا كتابة البرنامج بالشكل التالي:

```
x = 76
x = "Sammy"
print(x)
```

لن نتلقى سوى القيمة المسندة الثانية في المخرجات، لأنَّ تلك القيمة هي الأحدث:

```
Sammy
```

قد تكون إعادة إسناد القيم إلى المتغيرات مفيدة في بعض الحالات، لكن عليك أن تبقي عينك على مقروئية الشيفرة، وأن تحرص على جعل البرنامج واضحًا قدر الإمكان.

4. الإسناد المتعدد (Multiple Assignment)

في بايثون، يمكنك إسناد قيمة واحدة إلى عدة متغيرات في الوقت نفسه. يتيح لك هذا تهيئة عدَّة متغيرات دفعةً واحدةً، والتي يمكنك إعادة إسنادها لاحقًا، أو من خلال مدخلات المستخدم.

يمكنك من خلال الإسنادات المتعددة إسناد قيمة واحدة إلى عدَّة متغيرات (مثل المتغير x وy و z) في سطر واحد:

```
x = y = z = 0
print(x)
print(y)
print(z)
```

الناتج سيكون:

```
0
0
0
```

عرفنا في هذا المثال ثلاثة متغيرات (x و y و z)، وأسندنا إليها القيمة 0.

تسمح لك بايثون أيضًا بإسناد عدَّة قيم لعدَّة متغيِّرات ضمن السطر نفسه. هذه القيم يمكن أن تكون من أنواع بيانات مختلفة:

```
j, k, l = "shark", 2.05, 15
print(j)
print(k)
print(l)
```

وهذه هي المخرجات:

```
shark
2.05
15
```

في المثال أعلاه، أُسندَت السلسلة النصية "shark" إلى المتغير j، والعدد العشري 2.05 إلى المتغير k، والعدد الصحيح 15 إلى المتغير l.

إسناد عدة متغيرات بعدة قيم في سطر واحد يمكن أن يختصر الشيفرة ويقلل من عدد أسطرها، ولكن تأكد من أنَّ ذلك ليس على حساب المقروئية.

5. المتغيرات العامة والمحلية

عند استخدام المتغيرات داخل البرنامج، من المهم أن تضع نطاق (scope) المتغير في حساباتك. يشير نطاق المتغير إلى المواضع التي يمكن الوصول منها إلى المتغيِّر داخل الشيفرة، لأنَّه لا يمكن الوصول إلى جميع المتغيرات من جميع أجزاء البرنامج، فبعض المتغيرات عامة (global)، وبعضها محلي (local). مبدئيًّا، تُعرَّف المتغيرات العامة خارج الدوال؛ أمَّا المتغيرات المحلية، فتعرَّف داخل الدوال.

المثال التالي يعطي فكرة عن المتغيرات العامة والمحلية:

```
# إنشاء متغير عام، خارج الدالة
glb_var = "global"

def var_function():
    # إنشاء متغير محلي داخل دالة
    lcl_var = "local"
    print(lcl_var)

# استدعاء دالة لطباعة المتغير المحلي
var_function()

# طباعة متغير عام خارج دالة
print(glb_var)
```

المخرجات الناتجة:

```
local
global
```

يُسنِد البرنامج أعلاه سلسلة نصية إلى المتغير العمومي `glb_var` خارج الدالة، ثم يعرِّف الدالة `var_function()`. وسيُنشِئ داخل تلك الدالة متغيِّرًا محليًّا باسم `lcl_var`، ثم يطبعه قيمته. ينتهي البرنامج باستدعاء الدالة `var_function()`، وطباعة قيمة المتغير `glb_var`.

لمَّا كان `glb_var` متغيرًا عامًّا، فيمكننا الوصول إليه داخل الدالة `var_function()`. المثال التالي يبيِّن ذلك:

```
glb_var = "global"

def var_function():
    lcl_var = "local"
```

```
    print(lcl_var)
    print(glb_var) # طباعة glb_var داخل الدالة

var_function()
print(glb_var)
```

المخرجات:

```
local
global
global
```

لقد طبعنا المتغير العام `glb_var` مرتين، إذ طُبع داخل الدالة وخارجها.

ماذا لو حاولنا استدعاء المتغير المحلي خارج الدالة؟

```
glb_var = "global"

def var_function():
    lcl_var = "local"
    print(lcl_var)

print(lcl_var)
```

لا يمكننا استخدام متغير محلي خارج الدالة التي صُرِّح عنه فيها. إذا حاولنا القيام بذلك، فسيُطلق الخطأ `NameError`.

```
NameError: name 'lcl_var' is not defined
```

دعنا ننظر إلى مثال آخر، حيث سنستخدم الاسم نفسه لمتغير عام وآخر محلي:

```
num1 = 5 # متغير عام

def my_function():
    num1 = 10 # num1 استخدام نفس اسم المتغير
    num2 = 7  # تعيين متغير محلي

    print(num1) # num1 طباعة المتغير المحلي
    print(num2) # num2 طباعة المتغير المحلي

# my_function() استدعاء
my_function()

# num1 طباعة المتغير العام
print(num1)
```

الناتج:

```
10
7
5
```

نظرًا لأنَّ المتغير المحلي `num1` صُرِّح عنه محليًا داخل إحدى الدوال، فسنرى أنَّ `num1` يساوي القيمة المحلية 10 عند استدعاء الدالة. عندما نطبع القيمة العامة للمتغير `num1` بعد استدعاء الدالة `my_function()`، سنرى أنَّ المتغير العام `num1` لا يزال مساويًا للقيمة 5.

من الممكن الوصول إلى المتغيرات العامة واستعمالها داخل دالة باستخدام الكلمة المفتاحية `global`:

```
def new_shark():
    # جعل المتغير عاما
    global shark
    shark = "Sammy"

# new_shark() استدعاء الدالة
new_shark()

# shark طباعة المتغير العام
print(shark)
```

رغم أنَّ المتغير المحلي shark عُيِّن داخل الدالة new_shark()، إلا أنَّه يمكن الوصول إليه من خارج الدالة بسبب الكلمة المفتاحية global المستخدمة قبل اسم المتغير داخل الدالة.

بسبب استخدام global، فلن يُطلَق أيُّ خطأ عندما نستدعي print(shark) خارج الدالة. رغم أنَّه يمكنك استعمال متغير عام داخل دالة، إلا أنَّ ذلك يُعدُّ من العادات غير المستحبة في البرمجة، لأنَّها قد تؤثر على مقروئية الشيفرة عدا عن السماح لجزء من الشيفرة بتعديل قيمة متغير قد يستعمله جزء آخر.

هناك شيء آخر يجب تذكره، وهو أنَّك إذا أشرت إلى متغير داخل دالة، دون إسناد قيمة له، فسيُعَدّ هذا المتغير عامًا ضمنيًا. لجعل متغيرٍ محليًّا، يجب عليك إسناد قيمة له داخل متن الدالة.

عند التعامل مع المتغيرات، يكون لك الخيار بين استخدام المتغيرات العامة أو المحلية. يُفضَّل في العادة استخدام المتغيرات المحلية، ولكن إن وجدت نفسك تستخدم نفس المتغير في عدة دوال، فقد ترغب في جعله عامًا. أمَّا إن كنت تحتاج المتغير داخل دالة أو صنف واحد فقط، فقد يكون الأولى استخدام متغير محلي.

6. خلاصة الفصل

لقد مررنا في هذا الفصل على بعض حالات الاستخدام الشائعة للمتغيرات في بايثون 3. المتغيرات هي لبنة مهمة في البرمجة، إذ تُمثِّل حاضنةً لمختلف أنواع البيانات في بايثون والتي سنسلط عليها الضوء في الفصل التالي.

6

أنواع البيانات والتحويل بينها

تُصنِّف بايثون البيانات (data) إلى أنواع، كما هو الحال في جميع لغات البرمجة. هذا مهم لأنَّ نوع البيانات الذي تستخدمه سيقيِّد القيم التي يمكن تعيينها لها، وما الذي يمكن فعله بها (بما في ذلك العمليات التي يمكن تنفيذها عليها).

سنتعرَّف في هذا الفصل على أهم أنواع البيانات الأصليَّة لبايثون. هذا ليس استقصاءً شاملًا لأنواع البيانات، ولكنَّه سيساعدك على التعرف على الخيارات المتاحة لك في بايثون.

1. خلفية عامة

أنواع البيانات في بايثون مشابهة إلى حد ما لأنواع البيانات التي نستخدمها في العالم الحقيقي. من أمثلة أنواع البيانات في العالم الحقيقي الأعداد، مثل: الأعداد الصحيحة الطبيعية (0، 1، 2، …)، والأعداد الصحيحة النسبية (…، -1، 0، 1، …)، والأعداد غير النسبية (π).

يمكننا عادة في الرياضيات جمع أعداد من أنواع مختلفة مثل إضافة 5 إلى π:

```
5 + π
```

يمكننا إمّا الاحتفاظ بالمعادلة بِعدِّها إجابة، وستكون النتيجة عددًا غير نسبي (irrational number)، أو يمكننا تقريب (round) العدد π إلى عدد ذي منازل عشرية محددة، ثم نجمع العددين:

```
5 + π = 5 + 3.14 = 8.14
```

ولكن، إذا حاولنا إضافة عدد إلى نوع بيانات آخر، مثل الكلمات، فستصبح الأمور مربكة وغير ذات معنى. فكيف ستحل المعادلة التالية مثلًا؟

```
hsoub + 8
```

بالنسبة إلى الكلمة hsoub، يمكن عدُّ كل نوع من أنواع البيانات مختلفًا تمامًا، مثل الكلمات والأعـداد، لـذلك يتعيَّن علينـا تـوخي الحـذر بشـأن كيفيَّة اسـتخدامها، وكيفيَّة التعامـل معهـا في العمليات.

2. الأعداد

سيُفسَّر كل عدد تُدخله إلى بايثون على أنَّه عدد؛ ليس مطلوبًا منك الإعلان صراحةً عن نوع البيانات الذي تدخله لأنَّ بايثون تَعدُّ أيّ عدد مكتوب بدون فواصل عشرية بمثابة عدد صحيح (integer، كما هو حال 138)، وأيُّ عدد مكتوب بفواصل لعشرية بمثابة عدد عشري (float، كما هو حال 138.0).

ا. الأعداد الصحيحة (integer)

كما هو الحال في الرياضيات، الأعداد الصحيحة (integer) في البرمجة هي أعداد كاملة، يمكن أن تكون موجبة أو سالبة أو معدومة (…، 1-، 0، 1، …). ويُعرف هذا النوع أيضًا باسم int. كما هو الحال مع لغات البرمجة الأخرى، يجب ألا تستخدم الفواصل في الأعداد المؤلفة من أربعة أرقام أو أكثر، لذلك لا تكتب 1,000 في برنامجك، واكتب 1000.

يمكننا طباعة عدد صحيح على النحو التالي:

```
print(-25)
```

وسينتج:

```
-25
```

أو يمكننا الإعلان عن متغير، والذي هو في هذه الحالة رمزٌ للعدد الذي نستخدمه أو نتعامل معه، مثلا:

```
my_int = -25
print(my_int)
```

وسينتج لنا:

```
-25
```

يمكننا أن نجري العمليات الحسابية على الأعداد الصحيحة في بايثون مباشرةً مثل:

```
int_ans = 116 - 68
print(int_ans)
```

المخرجات:

```
48
```

يمكن استخدام الأعداد الصحيحة بعدة طرائق في برامج بايثون، ومع استمرارك في تعلم المزيد عن هذه اللغة، ستتاح لك الكثير من الفرص لاستخدام الأعداد الصحيحة والتعامل معها وفهم المزيد عن هذا النوع من البيانات.

ب. الأعداد العشرية (Floating-Point Numbers)

الأعداد العشرية هي أعداد حقيقية، مما يعني أنه يمكن أن تكون أعدادًا جذرية أو غير نسبية. لهذا السبب، يمكن أن تحتوي الأعداد العشرية على جزء كسري، مثل `9.0` أو `-116.42`. وببساطة، فالأعداد العشرية هي أعداد تحتوي الفاصلة العشرية.

كما فعلنا مع الأعداد الصحيحة، يمكننا طباعة الأعداد العشرية هكذا:

```
print(17.3)
```

وسينتج لنا:

```
17.3
```

يمكننا أيضًا أن نعلن عن متغيرٍّ يحوى عددًا عشريًا، مثلا:

```
my_flt = 17.3
print(my_flt)
```

الناتج:

```
17.3
```

وكما هو الحال مع الأعداد الصحيحة، يمكننا أن نجرى العمليات الحسابية على الأعداد العشرية:

```
flt_ans = 564.0 + 365.24
print(flt_ans)
```

الناتج:

```
929.24
```

تختلف الأعداد الصحيحة والأعداد العشرية عن بعضها عمومًا، إذ أنَّ 3 ≠ 3.0، لأنَّ 3 عدد صحيح، بينما 3.0 عدد عشري.

3. القيم المنطقية

هناك قيمتان فقط لنوع البيانات المنطقية (Boolean)، وهما True و False. تُستخدم القيم المنطقية لتمثيل قيم الحقيقة الموافقة للمنطق الرياضياتي (صح أو خطأ).

عادة ما يبدأ اسم البيانات المنطقية بالحرف B، إشارة إلى اسم عالم الرياضيات George Boole. القيمتان True و False تُكتبان دائمًا بحرفين كبيرين T و F، لأنَّها قيم خاصة في بايثون.

الكثير من العمليات الحسابية في الرياضيات تُنتج قيمًا منطقيًا، إما True أو False:

- أكبر من

```
bool_val = 500 > 100 # True
bool_val = 1 > 5     # False
```

- أصغر من

```
bool_val = 200 < 400 # True
bool_val = 4 < 2     # False
```

- التساوي

```
bool_val = 5 = 5     # True
bool_val = 500 = 400 # False
```

كما هو الحال مع الأعداد، يمكننا تخزين القيم المنطقية في المتغيرات:

```
my_bool = 5 > 8
```

يمكننا بعد ذلك طباعة القيمة المنطقية باستدعاء الدالة `print()`:

```
print(my_bool)
```

بما أنَّ العدد 5 ليس أكبر من 8، فسوف نحصل على المخرجات التالية:

```
False
```

ستتعلّم مع مرور الوقت كيفيّة استخدام القيم المنطقيّة، وكيف يمكن للدوال والعمليات المنطقيّة أن تغيِّر مسار البرنامج.

4. السلاسل النصية

السلسلة النصية (string) هي عبارة عن تسلسل من محرف واحد أو أكثر (محارف وأعداد ورموز)، ويمكن أن تكون ثابتة أو متغيرة. تحاط السلاسل النصية إما بعلامات الاقتباس المفردة ' أو علامات الاقتباس المزدوجة "، لذلك لإنشاء سلسلة نصية، ضع سلسلة من الأحرف بين علامتي اقتباس:

```
'This is a string in single quotes.'
"This is a string in double quotes."
```

يمكنك استخدام علامات الاقتباس المفردة أو علامات الاقتباس المزدوجة، المهم أن تكون متسقًا في برنامجك.

البرنامج البسيط "Hello, World!" يوضح كيف يمكن استخدام السلاسل النصية في البرمجة، إذ أنَّ حروف عبارة Hello, World! تمثل سلسلة نصية.

```
print("Hello, World!")
```

كما هو الحال مع أنواع البيانات الأخرى، يمكننا تخزين السلاسل النصية في المتغيرات:

```
hw = "Hello, World!"
```

وطباعة السلسلة عن طريق استدعاء المتغيِّر:

```
print(hw)  # Hello, World!
```

مثل الأعداد، هناك العديد من العمليات التي يمكن إجراؤها على السلاسل النصية من أجل تحقيق النتائج التي نسعى إليها. السلاسل النصية مهمة لتوصيل المعلومات إلى المستخدم، وكذلك لتمكين المستخدم من تمرير المعلومات إلى البرنامج.

5. القوائم (Lists)

القائمة (list) عبارة عن تسلسل مرتَّب قابل للتغيير (mutable). وكما تُعرَّف السلاسل النصيَّة باستخدام علامات الاقتباس، يتم تعريف القوائم باستخدام الأقواس المعقوفة `[]`.

مثلًا، هذه قائمة تحوي أعدادًا صحيحةً:

```
[-3, -2, -1, 0, 1, 2, 3]
```

وهذه قائمة من الأعداد العشرية:

```
[3.14, 9.23, 111.11, 312.12, 1.05]
```

وهذه قائمة من السلاسل النصية:

```
['shark', 'cuttlefish', 'squid', 'mantis shrimp']
```

في المثال التالي، سنسمِّي قائمة السلاسل النصية خاصتنا بالاسم `sea_creatures`:

```
sea_creatures = ['shark', 'cuttlefish', 'squid', 'mantis
shrimp']
```

يمكننا طباعتها عن طريق استدعاء المتغير:

```
print(sea_creatures)
```

وسترى أنَّ المخرجات تشبه تمامًا القائمة التي أنشأناها:

```
['shark', 'cuttlefish', 'squid', 'mantis shrimp']
```

القوائم هي نوع بيانات مرن للغاية، لأنها قابلة للتغيير، حيث يمكن إضافة قيم إليها، أو إزالته، أو تغييرها. هناك نوع بيانات آخر مشابه لقوائم، بيْد أنَّه غير قابل للتغيير، ويُسمى الصف (tuple).

6. الصفوف (Tuples)

يُستخدم الصف (tuple) لتجميع البيانات، وهو تسلسل ثابت من العناصر وغير قابل للتغيير. الصفوف تشبه القوائم إلى حد كبير، لكنها تستخدم الأقواس `()` بدلًا من الأقواس المعقوفة `[]`، ولأنَّها غير قابلة للتغيير، فلا يمكن تغيير أو تعديل قيمها.

تبدو الصفوف كالتالي:

```
('blue coral', 'staghorn coral', 'pillar coral')
```

يمكننا تخزين الصفوف في المتغيِّرات وطباعتها:

```
coral = ('blue coral', 'staghorn coral', 'pillar coral')
print(coral)
```

والمخرجات هي:

```
('blue coral', 'staghorn coral', 'pillar coral')
```

كما هو الحال في أنواع البيانات الأخرى، تطبع بايثون الصفوف تمامًا كما كتبناها، إذ تطبع سلسلة من القيم بين قوسين.

7. القواميس (Dictionaries)

القاموس (dictionary) هو نوع مُضمّن في بايثون، إذ تُربط مفاتيحُ بالقيم المقابلة لها في شكل أزواج، هذه الأزواج مفيدة لتخزين البيانات في بايثون. يتم إنشاء القواميس باستخدام الأقواس المعقوصة {}.

تُستخدم القواميس عادةً لحفظ البيانات المترابطة، مثل المعلومات المقابلة لرقم تعريف. يبدو القاموس كما يلي:

```
{'name': 'Sammy', 'animal': 'shark', 'color': 'blue',
'location': 'ocean'}
```

ستلاحظ أنه بالإضافة إلى الأقواس المعقوصة، توجد علامات النقطتين الرأسيتين (colons) داخل القاموس. الكلمات الموجودة على يسار النقطتين الرأسيتين هي المفاتيح. المفاتيح قد تكون أيَّ نوع بيانات غير قابل للتغيير. المفاتيح في القاموس أعلاه هي:

```
'name', 'animal', 'color', 'location'
```

الكلمات الموجودة على يمين النقطتين هي القيم. يمكن أن تتألف القيم من أي نوع من البيانات. القيم في القاموس أعلاه هي:

```
'Sammy', 'shark', 'blue', 'ocean'
```

مثل أنواع البيانات الأخرى، يمكننا تخزين القواميس في متغيرات، وطباعتها:

```
sammy = {'name': 'Sammy', 'animal': 'shark', 'color': 'blue',
'location': 'ocean'}
print(sammy)
```

والمخرجات هي:

```
{'color': 'blue', 'animal': 'shark', 'name': 'Sammy',
'location': 'ocean'}
```

إذا أردت الحصول على اللون (color) الخاص بـ Sammy، فيمكنك القيام بذلك عن طريق استدعاء `sammy ['color']`. هذا مثال على ذلك:

```
print(sammy['color']) # blue
```

القواميس من أنواع البيانات المهمّة في برامج بايثون.

8. التحويل بين أنواع البيانات

يحدِّد نوع البيانات -كما ذكرنا- القيم التي يمكن استعمالها، والعمليات التي يمكنك إجراؤها عليها. هناك أوقات نحتاج إلى تحويل القيم من نوعٍ إلى آخر لأجل معالجتها بطريقة مختلفة. على سبيل المثال، قد نحتاج إلى ضم (concatenate) القيم العددية إلى سلاسل نصية.

سيرشدك هذا القسم إلى كيفية التحويل بين الأعداد والسلاسل النصية والصفوف والقوائم، بالإضافة إلى تقديم بعض الأمثلة التوضيحية.

ا. تحويل الأنواع العددية

هناك نوعان من البيانات العددية في بايثون كما رأينا آنفًا: الأعداد الصحيحة والأعداد العشرية. ستعمل في بعض الأحيان على شيفرة برمجية كتبها شخص آخر، وقد تحتاج إلى تحويل عدد صحيح إلى عدد عشري، أو العكس، أو قد تجد أنَّك تستخدم عددًا صحيحًا في الوقت الذي تحتاج إلى أعداد عشرية. يتوفر في بايثون توابع مضمّنة تُسهِّل عليك تحويل الأعداد الصحيحة إلى أعداد عشريَّة، أو العكس.

تحويل الأعداد الصحيحة إلى أعداد عشرية

يحوّل التابع `float()` الأعداد الصحيحة إلى أعداد عشرية. لاستخدام هذه الدالة، ضع عددًا صحيحًا بين القوسين:

```
float(57)
```

في هذه الحالة، سيحوَّل العدد الصحيح 57 إلى العدد العشري 57.0.

يمكنك أيضًا استخدام هذه الدالة مع المتغيرات. لنُسنِد القيمة 57 إلى المتغيِّر f، ثم نطبع العدد العشري الجديد:

```
f = 57

print(float(f))
```

الناتج سيكون:

```
57.0
```

يمكننا باستخدام الدالة `float()` تحويل الأعداد الصحيحة إلى أعداد عشرية.

تحويل الأعداد العشرية إلى أعداد صحيحة

تملك بايثون دالة أخرى مضمَّنة لتحويل الأعداد عشرية إلى أعداد صحيحة: وهي `int()`.

تعمل الدالة `int()` بشكل مشابه للدالة `float()`: يمكنك إضافة عدد عشري داخل القوسين لتحويله إلى عدد صحيح:

```
int(390.8)
```

في هذه الحالة، سيُحوَّل العدد العشري 390.8 إلى العدد الصحيح 390.

يمكنك أيضًا استخدام هذه الدالة مع المتغيرات. لنصرّح أنَّ b يساوي 125.0، وأنَّ c يساوي 390.8، ثم نطبع العددين العشريين الجديدين:

```
b = 125.0

c = 390.8



print(int(b))

print(int(c))
```

والمخرجات ستكون:

```
125
390
```

عند تحويل الأعداد العشرية إلى أعداد صحيحة بواسطة الدالة int()، تقتطع بايثون الأجزاء العشرية من العدد وتُبقي القيمة الصحيحة؛ لذلك، لن تُحوِّل الدالة int() العدد 390.8 إلى 391.

تحويل الأعداد عبر القسمة

في بايثون 3، عند تقسيم عدد صحيح على آخر، سينتج عدد عشري على خلاف بايثون 2. بمعنى أنَّه عند قسمة 5 على 2 في بايثون 3، ستحصل على عدد عشري (مثل 2.5 عند قسمة 5 على 2):

```
a = 5 / 2

print(a)
```

وسينتج لنا:

```
2.5
```

بينما في بايثون 2، ستحصل على ناتج صحيح، أي `5/2 = 2`. يمكنك الحصول على عدد صحيح ناتج عن عملية القسمة باستعمال المعامل `//` الجديد في بايثون 3:

```
a = 5 // 2

print(a)
```

وسينتج لنا:

```
2
```

ارجع إلى فصل «إصدارات بايثون: بايثون 2 مقابل بايثون 3» للاطلاع على المزيد من الفروقات بين بايثون 2 وبايثون 3.

ب. التحويل مع السلاسل النصية

السلاسل النصية عبارة عن سلسلة مؤلفة من محرف واحد أو أكثر (المحرف يمكن أن يكون حرفًا، أو عددًا، أو رمزًا). السلاسل النصية هي إحدى الأشكال الشائعة من البيانات في عالم البرمجة، وقد نحتاج إلى تحويل السلاسل النصية إلى أعداد أو أعداد إلى سلاسل نصية في كثير من الأحيان، خاصةً عندما نعمل على البيانات التي ينشئها المستخدمون.

تحويل الأعداد إلى سلاسل نصية

يمكننا تحويل الأعداد إلى سلاسل نصية عبر التابع `str()`. يمكننا أن نمرِّر إمَّا عددًا أو متغيرًا بين قوسي التابع، وبعد ذلك ستُحوَّل تلك القيمة العددية إلى قيمة نصية.

دعنا ننظر أولًا في تحويل الأعداد الصحيحة. لتحويل العدد الصحيح 12 إلى سلسلة نصية، يمكنك تمرير 12 إلى التابع `str()`:

```
str(12)
```

عند تنفيذ `str(12)` في سطر أوامر بايثون التفاعلي مع الأمر `python` في نافذة الطرفية، ستحصل على المخرجات التالية:

```
'12'
```

تشير علامات الاقتباس المحيطة بالعدد 12 إلى أنه لم يعد عددًا صحيحًا، ولكنَّه أصبح الآن سلسلة نصية.

سيصبح باستخدام المتغيرات تحويل الأعداد الصحيحة إلى سلاسل نصية أكثر فائدة. لنفترض أنَّنا نريد متابعة تقدُّم مستخدم في مجال البرمجة يوميًّا مثل أن ندخل عدد أسطر الشيفرة البرمجية التي كتبها. نود أن نعرض ذلك على المستخدم، وذلك بطباعة السلاسل النصية والأعداد في الوقت نفسه:

```
user = "Sammy"

lines = 50

print("Congratulations, " + user + "! You just wrote " + lines
+ " lines of code.")
```

عند تنفيذ الشيفرة أعلاه، سيُطلَق الخطأ التالي:

```
TypeError: Can't convert 'int' object to str implicitly
```

يتعذر علينا ضمُّ (concatenate) الأعداد إلى السلاسل النصية في بايثون، لذلك يجب تحويل المتغير `lines` إلى سلسلة نصية:

```
user = "Sammy"
lines = 50

print("Congratulations, " + user + "! You just wrote " +
str(lines) + " lines of code.")
```

الآن، عندما نُنفِّذ الشيفرة البرمجيَّة، سنحصل على المخرجات التالية، وفيها تهنئة للمستخدم على تقدُّمه:

```
Congratulations, Sammy! You just wrote 50 lines of code.
```

إذا أردنا تحويل عدد عشري إلى سلسلة نصية بدلًا من تحويل عدد صحيح إلى سلسلة نصية، فعلينا تتبع نفس الخطوات والصياغة السابقة. عندما نمرِّر عددًا عشريًا إلى التابع `str()`، ستُعاد سلسلة نصية. يمكننا استخدام قيمة العدد العشري نفسها، أو يمكننا استخدام متغيِّر:

```
print(str(421.034))

f = 5524.53
print(str(f))
```

وسينتج لنا:

```
421.034
5524.53
```

يمكننا اختبار صحة التحويل عن طريق ضم الناتج إلى سلسلة نصية:

```
f = 5524.53

print("Sammy has " + str(f) + " points.")
```

وهذا هو الناتج:

```
Sammy has 5524.53 points.
```

الآن تأكدنا من أنَّ عددنا العشري قد حُوِّل بنجاح إلى سلسلة نصية، لأنَّ عملية الضم قد نُفِّذت دون خطأ.

تحويل السلاسل النصية إلى أعداد

يمكن تحويل السلاسل النصية إلى أعداد باستخدام التابعين `float()` و `int()`. إذا لم يكن في السلسلة النصية منازل عشرية، فالأفضل أن تحولها إلى عدد صحيح باستخدام التابع `int()`.

دعنا نستخدم مثال تتبع عدد أسطر الشيفرة الذي أوردناه أعلاه. قد ترغب في التعامل مع هذه القيم باستخدام الحسابات الرياضياتية لتقديم نتائج أدق للمستخدم، ولكنَّ هذه القيم مخزَّنة حاليًا في سلاسل نصية:

```
lines_yesterday = "50"

lines_today = "108"

lines_more = lines_today - lines_yesterday

print(lines_more)
```

الناتج هو:

```
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

نظرًا لأنَّ القيمتين العدديتين مخزَّنتان في سلاسل نصية، تلقينا خطأً. سبب ذلك أنَّ معامل الطرح - لا يصلح للسلاسل النصية.

دعنا نعدِّل الشيفرة لتضمين التابع `int()` الذي سيحول السلاسل النصية إلى أعداد صحيحة، ويسمح لنا بالقيام بالعمليات الرياضياتية على القيم التي كانت سلاسل نصية في الأصل.

```
lines_yesterday = "50"

lines_today = "108"

lines_more = int(lines_today) - int(lines_yesterday)

print(lines_more)
```

وهذه هي المخرجات:

```
58
```

المتغير `line_more` هو عدد صحيح تلقائيًّا، ويساوي القيمة العددية 58 في هذا المثال.

يمكننا أيضًا تحويل الأعداد في المثال أعلاه إلى قيم عشرية باستخدام التابع `float()` بدلًا من التابع `int()`. وبدلاً من الحصول على الناتج 58، سنحصل على الناتج 58.0، وهو عدد عشري.

سيكسب المستخدم Sammy نقاطًا على شكل قيم عشرية:

```
total_points = "5524.53"

new_points = "45.30"

new_total_points = total_points + new_points

print(new_total_points)
```

الناتج:

```
5524.5345.30
```

في هذه الحالة، يعدُّ استخدام المعامل + مع سلسلتين نصيتين عمليةً صالحةً، لكنه سيضم السلسلتين النصّيتين بدلًا من جمع القيمتين العدديتين؛ لذلك، سيبدو الناتج غير مألوف، لأنَّه نتيجة لصق القيمتين إلى جانب بعضهما بعضًا.

سنحتاج إلى تحويل هذه السلاسل النصية إلى أعداد عشرية قبل إجراء أي عمليات عليها، وذلك باستخدام التابع `float()`:

```
total_points = "5524.53"

new_points = "45.30"

new_total_points = float(total_points) + float(new_points)

print(new_total_points)
```

وسينتج عن ذلك:

```
5569.83
```

الآن، وبعد أن حوَّلنا السلسلتين النصيتين إلى عددين عشريين، سنحصل على النتيجة المتوقعة، والتي هي جمع 45.30 و 5524.53.

إذا حاولنا تحويل سلسلة نصية ذات منازل عشرية إلى عدد صحيح، فسنحصل على خطأ:

```
f = "54.23"

print(int(f))
```

المخرجات:

```
ValueError: invalid literal for int() with base 10: '54.23'
```

إذا مرّرنا عددًا عشريًا موضوعًا في سلسلة نصية إلى التابع `int()`، فسنحصل على خطأ، إذ لن تُحوَّل إلى عدد صحيح.

يتيح لنا تحويل السلاسل النصية إلى أعداد تعديل نوع البيانات الذي نعمل عليه بسرعة حتى نتمكن من إجراء عمليات على قيم عددية مكتوبة على شكل سلاسل نصية.

ج. التحويل إلى صفوف وقوائم

يمكنك استخدام التابعين `list()` و `tuple()` لتحويل القيم المُمرَّرة إليهما إلى قائمة أو صف على التوالي. في بايثون:

- القائمة هي تسلسل مرتَّب قابل للتغيير من العناصر الموضوعة داخل قوسين معقوفين `[]`.
- الصف عبارة عن تسلسل مرتب ثابت (غير قابل للتغيير) من العناصر الموضوعة بين القوسين الهلاليّين `()`.

التحويل إلى صفوف

نظرًا لكون الصفوف غير قابلة للتغيير، فيمكن أن يحسّن تحويل قائمة إلى صف أداء البرامج تحسينًا كبيرًا. عندما نستخدم التابع `tuple()`، فسوف يُعيد القيمة المُمرَّرة إليه على هيئة صف.

```
print(tuple(['pull request', 'open source', 'repository', 'branch']))
```

المخرجات:

```
('pull request', 'open source', 'repository', 'branch')
```

نرى أنَّ الصف قد طُبع في المخرجات، إذ أنَّ العناصر موضوعة الآن بين قوسين، بدلًا من القوسين المربعين.

دعنا نستخدم `tuple()` مع متغير يحتوي قائمة:

```
sea_creatures = ['shark', 'cuttlefish', 'squid', 'mantis shrimp']

print(tuple(sea_creatures))
```

سينتج:

```
('shark', 'cuttlefish', 'squid', 'mantis shrimp')
```

مرة أخرى، نرى أنَّ القائمة حُوِّلت إلى صف، كما يشير إلى ذلك القوسان. يمكننا تحويل أي نوع قابل للتكرار (iterable) إلى صف، بما في ذلك السلاسل النصية:

```
print(tuple('Sammy'))
```

المخرجات:

```
('S', 'a', 'm', 'm', 'y')
```

لمّا كان بالإمكان المرور (iterate) على محارف السلاسل النصية، فيمكننا تحويلها إلى صفوف باستخدام التابع `tuple()`. أمّا أنواع البيانات غير القابلة للتكرار، مثل الأعداد الصحيحة والأعداد العشرية، فستُطلِق عملية تحويلها خطأً:

```
print(tuple(5000))
```

والناتج سيكون:

```
TypeError: 'int' object is not iterable
```

في حين أنه من الممكن تحويل عدد صحيح إلى سلسلة نصيَّة، ومن ثم تحويل السلسلة النصية إلى صف، كما في `tuple(str(5000))`، فمن الأفضل تجنب مثل هذه التعليمات البرمجية المعقَّدة.

التحويل إلى قوائم

يمكن أن يكون تحويل القيم، وخاصة الصفوف، إلى قوائم مفيدًا عندما تحتاج إلى نسخة قابلة للتغيير من تلك القيم.

سنستخدم التابع `list()` لتحويل الصف التالي إلى قائمة. ونظرًا لأنَّ صياغة القوائم تستخدم الأقواس، تأكد من تضمين أقواس التابع `list()`، وكذلك الأقواس الخاصة بالدالة `print()`:

```
print(list(('blue coral', 'staghorn coral', 'pillar coral')))
```

المخرجات هي:

```
['blue coral', 'staghorn coral', 'pillar coral']
```

تشير الأقواس المعقوفة [] إلى أنَّه قد أُرجِعَت قائمة من الصف الأصلي الذي مُرِّر عبر الدالة list().

لجعل الشيفرة سهلة القراءة، يمكننا إزالة أحد أزواج الأقواس باستخدام متغيِّر:

```
coral = ('blue coral', 'staghorn coral', 'pillar coral')

list(coral)
```

إن طبعنا list(coral)، فسنتلقى المخرجات نفسها الموجودة أعلاه.

تمامًا مثل الصفوف، يمكن تحويل السلاسل النصية إلى قوائم:

```
print(list('shark'))
```

الناتج:

```
['s', 'h', 'a', 'r', 'k']
```

حُوِّلَت هنا السلسلة shark إلى قائمة، وهذا يوفِّر لنا نسخة قابلة للتغيير من القيمة الأصلية.

9. خلاصة الفصل

في هذه المرحلة، يُفترض أن يكون لديك فهم جيد لبعض أنواع البيانات الرئيسية المتاحة في بايثون. أنواع البيانات هذه ستصبح جزءًا طبيعيًا من حياتك كمبرمج للغة بايثون.

لقد وضحنا أيضًا في هذا الفصل كيفيَّة تحويل العديد من أنواع البيانات الأصلية المهمَّة إلى أنواع بيانات أخرى، وذلك باستخدام التوابع المُضمّنة. يوفر تحويل أنواع البيانات في بايثون لك

مرونةً إضافيةً في مشاريعك البرمجية. يمكنك التعرف على المزيد من التفاصيل عن هذه الأنواع وطرائق التحويل بينها في موسوعة حسوب.

7

السلاسل النصية والتعامل معها

«السلسلة النصية» (string) هي مجموعة من المحارف (أي الأحرف والأرقام والرموز) التي إمَّا أن تكون قيمة ثابتة أو قيمة لمتغير. وهذه السلاسل النصية مُشكَّلة من محارف يونيكود (Unicode) والتي لا يمكن تغيير مدلولها. ولأنَّ النص هو شكلٌ شائعٌ من أشكال البيانات الذي نستعمله يوميًا، لذا فإنَّ السلاسل النصية مهمة جدًا وتُمثِّل لُبنةً أساسيةً في البرمجة. سيستعرض هذا الفصل كيفيَّة إنشاء وطباعة السلاسة النصية، وكيفية جمعها مع بعضها وتكرارها، وآلية تخزين السلاسل النصية في متغيرات.

1. إنشاء وطباعة السلاسل النصية

تتواجد السلاسل النصية إمَّا داخل علامات اقتباس فردية ' أو علامات اقتباس مزدوجة "، لذا لإنشاء سلسلة نصية، كل ما علينا فعله هو وضع مجموعة من المحارف بين أحد نوعَي علامات الاقتباس السابقَين:

```
'هذه سلسلة نصية ضمن علامتي اقتباس مفردتين.'
"هذه سلسلة نصية ضمن علامتي اقتباس مزدوجتين"
```

يمكنك الاختيار بين النوعَين السابقَين، لكن أيًّا كان اختيارك، فعليك أن تحافظ على استخدامك له في كامل برنامجك. يمكنك طباعة السلاسل النصية إلى الشاشة باستدعاء الدالة `print()` بكل بساطة:

```
print("Let's print out this string.")
Let's print out this string.
```

بعد أن فهمتَ كيفية تهيئة السلاسل النصية في بايثون، لنلقِ نظرةً الآن إلى كيفيَّة التعامل مع السلاسل النصية في برامجك وتعديلها.

2. آلية فهرسة السلاسل النصية

وكما في نوع البيانات `list` الذي فيه عناصر مرتبطة بأرقام، فإنَّ كل محرف في السلسلة النصية يرتبط بفهرس معيّن، بدءًا من الفهرس 0. فللسلسلة النصية `Sammy Shark!` ستكون الفهارس والمحارف المرتبطة بها كالآتي:

s	a	m	m	y		s	h	a	r	k	!
0	1	2	3	4	5	6	7	8	9	10	11

وكما لاحظت، سيرتبط المحرف `S` بالفهرس 0، وستنتهي السلسلة النصية بالفهرس `11` مع الرمز `!`. لاحظ أيضًا أنَّ الفراغ بين كلمتَي `Sammy` و `Shark` له فهرسٌ خاصٌ به، وفي مثالنا سيكون له الفهرس `5`. علامة التعجب `!` لها فهرسٌ خاصٌ بها أيضًا، وأيَّة رموز أو علامات ترقيم أخرى مثل `*#$&.;?` سترتبط بفهرسٍ مخصص لها. حقيقة أنَّ المحارف في بايثون لها فهرس خاص بها ستعني أنَّ بإمكاننا الوصول إلى السلاسل النصية وتعديلها كما نفعل مع أنواع البيانات المتسلسلة الأخرى.

ا. الوصول إلى المحارف بفهارس موجبة

يمكننا الحصول على محرف من سلسلة نصية بالإشارة إليه عبر فهرسه. يمكننا فعل ذلك بوضع رقم الفهرس بين قوسين معقوفين `[]`. سنُعرِّف في المثال الآتي سلسلةً نصيةً ونطبع المحرف المرتبط بالفهرس المذكور بين قوسين:

```
ss = "Sammy Shark!"
print(ss[4])
```

الناتج:

```
y
```

عندما نُشير إلى فهرسٍ معيّنٍ في سلسلةٍ نصيةٍ، فستُعيد بايثون المحرف الموجود في ذاك الموضع، ولمّا كان المحرف y موجودًا في الفهرس الرابع في السلسلة النصية "Sammy Shark!" فعندما طبعنا ss[4] فظهر الحرف y.

خلاصة ما سبق، أرقام الفهارس ستسمح لنا بالوصول إلى محارف معيّنة ضمن سلسلة نصية.

ب. الوصول إلى المحارف بفهارس سالبة

إذا كانت لديك سلسلةٌ طويلةٌ وأردنا تحديد أحد محارفها لكن انطلاقًا من نهايتها، فعندئذٍ نستطيع استخدام الأرقام السالبة للفهارس، بدءًا من الفهرس 1-. لو أردنا أن نستخدم الفهارس السالبة مع السلسلة النصية Sammy Shark!، فستبدو كما يلي:

s	a	m	m	y		s	h	a	r	k	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

يمكننا أن نطبع المحرف r في السلسلة السابقة في حال استخدمنا الفهارس السالبة بالإشارة إلى المحرف الموجود في الفهرس 3- كما يلي:

```
print(ss[-3])
```

وجدنا أنَّه يمكننا الاستفادة من الفهارس السالبة لو أردنا الوصول إلى محرف في آخر سلسلة نصية طويلة.

3. تقسيم السلاسل النصية

يمكننا أن نحصل على مجال من المحارف من سلسلة نصية، فلنقل مثلًا أننا نريد أن نطبع الكلمة Shark فقط، يمكننا فعل ذلك بإنشائنا «لقسم» من السلسلة النصية، والذي هو سلسلةٌ من المحارف الموجودة ضمن السلسلة الأصلية. فالأقسام تسمح لنا بالوصول إلى عدّة محارف دفعةً واحدة باستعمال مجال من أرقام الفهارس مفصولة فيما بينها بنقطتين رأسيتين [x:y]:

```
print(ss[6:11])
```

الناتج:

```
Shark
```

عند إنشائنا لقسم مثل [6:11] فسيُمثِّل أوّل رقم مكان بدء القسم (متضمنًا المحرف الموجود عند ذاك الفهرس)، والرقم الثاني هو مكان نهاية القسم (دون تضمين ذاك المحرف)، وهذا هو السبب وراء استخدمنا لرقم فهرس يقع بعد نهاية القسم الذي نريد اقتطاعه في المثال السابق. نحن نُنشِئ « سلسلةً نصيةً فرعيةً» (substring) عندما نُقسِّم السلاسل النصية، والتي هي سلسلةٌ موجودةٌ ضمن سلسلةٍ أخرى. وعندما نستخدم التعبير ss[6:11] فنحن نستدعي السلسلة النصية Shark التي تتواجد ضمن السلسلة النصية Sammy Shark!. إذا أردنا تضمين نهاية السلسلة (أو بدايتها) في القسم الذي ستُنشِئه، فيمكن ألّا نضع أحد أرقام الفهارس في string[n:n]. فمثلًا، نستطيع أن نطبع أوّل كلمة من السلسلة ss - أي Sammy - بكتابة ما يلي:

```
print(ss[:5])
```

فعلنا ذلك بحذف رقم الفهرس قبل النقطتين الرأسيتين، ووضعنا رقم فهرس النهاية فقط، الذي يُشير إلى مكان إيقاف اقتطاع السلسلة النصية الفرعيَّة. لطباعة منتصف السلسلة النصية إلى آخرها، فسنضع فهرس البداية فقط قبل النقطتين الرأسيتين، كما يلي:

```
print(ss[7:])
```

الناتج:

```
hark!
```

بكتابة فهرس البداية فقط قبل النقطتين الرأسيتين وترك تحديد الفهرس الثاني، فإنَّ السلسلة الفرعية ستبدأ من الفهرس الأول إلى نهاية السلسلة النصية كلها. يمكنك أيضًا استخدام الفهارس السالبة في تقسيم سلسلة نصية، فكما ذكرنا سابقًا، تبدأ أرقام الفهارس السلبية من الرقم ‎-1، ويستمر العد إلى أن نصل إلى بداية السلسلة النصية. وعند استخدام الفهارس السالبة فسنبدأ من الرقم الأصغر لأنَّه يقع أولًا في السلسلة. لنستخدم فهرسين ذوي رقمين سالبين لاقتطاع جزء من السلسلة النصية ss:

```
print(ss[-4:-1])
```

الناتج:

```
ark
```

السلسلة النصية "ark" مأخوذة من السلسلة النصية "Sammy Shark!" لأنَّ الحرف a يقع في الموضع ‎-4 والحرف k يقع قبل الفهرس ‎-1 مباشرةً.

يمكن تحديد الخطوة عند تقسيم السلاسل النصية وذلك بتمرير معامل ثالث إضافةً إلى فهرسَي البداية والنهاية، وهو الخطوة، التي تُشير إلى عدد المحارف التي يجب تجاوزها بعد

الحصول على المحرف من السلسلة النصية. لم نُحدِّد إلى الآن الخطوة في أمثلتنا، إلا أنَّ قيمته الافتراضية هي 1، لذا سنحصل على كل محرف يقع بين الفهرسين. لننظر مرةً أخرى إلى المثال السابق الذي يطبع السلسلة النصية الفرعية "Shark":

```
print(ss[6:11])  # Shark
```

سنحصل على نفس النتائج بتضمين معامل ثالث هو الخطوة وقيمته 1:

```
print(ss[6:11:1])  # Shark
```

إذًا، إذا كانت الخطوة 1 فهذا يعني أنَّ بايثون ستُضمِّن جميع المحارف بين فهرسين، وإذا حذفتَ الخطوة فستعدُّها بايثون مساويةً للواحد. أما لو زدنا الخطوة، فسنرى أنَّ بعض المحارف ستهمل:

```
print(ss[0:12:2])  # SmySak
```

تحديد الخطوة بقيمة 2 كما في `ss[0:12:2]` سيؤدي إلى تجاوز حرف بين كل حرفين، ألقِ نظرةً على المحارف المكتوبة بخط عريض:

```
Sammy Shark!
```

لاحظ أنَّ الفراغ الموجود في الفهرس 5 قد أُهمِل أيضًا عندما كانت الخطوة 2. إذا وضعنا قيمةً أكبر للخطوة، فسنحصل على سلسلةٍ نصيةٍ فرعيةٍ أصغر بكثير:

```
print(ss[0:12:4])  # Sya
```

حذف الفهرسين وترك النقطتين الرأسيتين سيؤدي إلى إبقاء كامل السلسلة ضمن المجال، لكن إضافة معامل ثالث وهو الخطوة سيُحدِّد عدد المحارف التي سيتم تخطيها. إضافةً إلى ذلك، يمكنك تحديد رقم سالب كخطوة، مما يمكّنك من كتابة السلسلة النصية بترتيبٍ معكوس إذا

استعملتَ القيمة ‎-1‎:

```
print(ss[::-1])
```

الناتج:

```
!krahS ymmaS
```

لنجرِّب ذلك مرةً أخرى لكن إذا كانت الخطوة ‎-2‎:

```
print(ss[::-2])
```

الناتج:

```
!rh ma
```

في المثال السابق (`ss[::-2]`)، سنتعامل مع كامل السلسلة النصية لعدم وجود أرقام لفهارس البداية والنهاية، وسيتم قلب اتجاه السلسلة النصية لاستخدامنا لخطوةٍ سالبة. بالإضافة إلى أنَّ الخطوة ‎-2‎ ستؤدي إلى تخطي حرف بين كل حرفين بترتيبٍ معكوس:

```
!krahS[whitespace]ymmaS
```

سيُطبَع الفراغ في المثال السابق.

ما رأيناه هو أنَّ تحديد المعامل الثالث عند تقسيم السلاسل النصية سيؤدي إلى تحديد الخطوة التي تُمثِّل عدد المحارف التي سيتم تخطيها عند الحصول على السلسلة الفرعية من السلسلة الأصلية.

4. جمع السلاسل النصية

عمليـة الجمع (concatenation) تعني إضـافة سلسـلتين نصيتين إلى بعضـهما بعضًا لإنشـاء سلسـلة نصـية جديـدة. نسـتخدم المعامـل + لجمـع السلاسـل النصـية؛ أبـقِ في ذهنـك أنَّ العامـل + يعـني عمليـة الجمـع عنـد التعامـل مـع الأعـداد، أمـا عنـدما نسـتخدمه مـع السلاسـل النصـية فيعـني إضـافتها إلى بعضـها. لنجمـع السلسـتين النصـيتين "Sammy" و "Shark" مـع بعضـها ثم نطبعهما باستخدام الدالة print()‎:

```
print("Sammy" + "Shark")
# SammyShark
```

إذا أردتَ وضـع فـراغ بين السلسـلتين النصـيتين، فيمكنـك بكـل بسـاطة وضـعه عنـد نهايـة السلسلة النصية الأولى، أي بعد الكلمة "Sammy":

```
print("Sammy " + "Shark")
# Sammy Shark
```

لكن احـرص على عدم اسـتعمال العامل + بين نـوعَين مختلفَين من البيانات، فلن نتمكن من جمع السلاسل النصية والأرقام مع بعضها، فلو حاولنا مثلًا أن نكتب:

```
print("Sammy" + 27)
```

فسنحصل على رسالة الخطأ الآتية:

```
TypeError: Can't convert 'int' object to str implicitly
```

أمَّا إذا أردنـا أن نُنشِـئ السلسـلة النصـية "Sammy27" فعلينـا حينهـا وضـع الـرقم 27 بين علامتَي اقتباس ("27") مما يجعله سلسلةً نصيةً وليست عددًا صحيحًا. سنستفيد من تحويل الأعداد إلى سلاسل نصية عندما نتعامل مع أرقام الهواتف على سبيل المثال، لأنَّنا لن نحتاج إلى

إجراء عملية حسابية على رمز الدولة ورمز المنطقة في أرقام الهواتف، إلا أنَّنا نريدهما أن يظهرا متتابعَين. عندما نجمع سلسلتين نصيتين أو أكثر فنحن نُنشِئ سلسلةً نصيةً جديدةً التي يمكننا استخدامها في برنامجنا.

5. تكرار السلاسل النصية

هنالك أوقاتٌ نحتاج فيها إلى استخدام بايثون لأتمتة المهام، وإحدى الأمور التي يمكننا أتمتتها هي تكرار سلسلة نصية لعدّة مرات. إذ نستطيع فعل ذلك عبر العامل *، وكما هو الأمر مع العامل + فإنَّ العامل * له استخدامٌ مختلف عندما نتعامل مع أرقام، حيث يُمثِّل عملية الضرب. أمَّا عندما نستخدمه بين سلسلةٍ نصيةٍ ورقمٍ فإنَّ العامل * هو معامل التكرار، فوظيفته هي تكرار سلسلة نصية لأي عدد مرات تشاء. لنحاول طباعة السلسلة النصية "Sammy" تسع مرات دون تكرارها يدويًا، وذلك عبر العامل *:

```
print("Sammy" * 9)
```

المخرجات:

```
SammySammySammySammySammySammySammySammySammy
```

يمكننا بهذه الطريقة تكرار السلسلة النصية لأيِّ عددٍ نشاء من المرات.

6. تخزين السلاسل النصية في متغيرات

وجدنا من فصل المتغيِّرات أنَّ المتغيِّرات هي «رموز» يمكننا استعمالها لتخزين البيانات في برنامج. أي يمكنك تخيل المتغيِّرات على أنَّها صندوقٌ فارغٌ يمكنك ملؤه بالبيانات أو القيم. السلاسل النصية هي نوعٌ من أنواع البيانات، لذا يمكننا استعمالها لملء المتغيرات. التصريح عن

متغيرات تحوي سلاسل نصية سيُسهِّل علينا التعامل معها في برامجنا. لتخزين سلسلة نصية داخل متغير، فكل ما علينا فعله هو إسنادها إليه. سنُصرِّح في المثال الآتي عن المتغير my_str:

```
my_str = "Sammy likes declaring strings."
```

أصبح المتغير my_str الآن مُشيرًا إلى سلسلةٍ نصيةٍ، والتي أمسى بمقدورنا طباعتها كما يلي:

```
print(my_str)
```

وسنحصل على الناتج الآتي:

```
Sammy likes declaring strings.
```

استخدام المتغيرات لاحتواء قيم السلاسل النصية سيساعدنا في الاستغناء عن إعادة كتابة السلسلة النصية في كل مرة نحتاج استخدامها، مما يُبسِّط تعاملنا معها وإجراءنا للعمليات عليها في برامجنا.

7. دوال السلاسل النصية

لدى بايثون عدّة دوال مبنية فيها للتعامل مع السلاسل النصية. تسمح هذه الدوال لنا بتعديل وإجراء عمليات على السلاسل النصية بسهولة. يمكنك أن تتخيل الدوال على أنَّها «أفعال» يمكننا تنفيذها على عناصر موجودة في الشيفرة. الدوال المبنية في اللغة هي الدوال المُعرَّفة داخل لغة بايثون وهي جاهزة مباشرةً للاستخدام. سنشرح في هذا القسم مختلف الدوال التي نستطيع استخدامها للتعامل مع السلاسل النصية في بايثون 3.

ا. جعل السلاسل النصية بأحرف كبيرة أو صغيرة

الدالتان `str.upper()` و `str.lower()` ستُعيدان السلسلة النصية بعد تحويل حالة جميع أحرفها الأصليَّة إلى الأحرف الكبيرة أو الصغيرة (على التوالي وبالترتيب). ولعدم قدرتنا على تعديل السلسلة النصية بعد إنشائها، فستُعاد سلسلةٌ نصيةٌ جديدةٌ. لن تُعدَّل أيَّة محارف غير لاتينية في السلسلة النصية الأصلية وستبقى على حالها. لنحوِّل السلسلة النصية `Sammy Shark` إلى أحرفٍ كبيرةٍ:

```
ss = "Sammy Shark"
print(ss.upper())
```

الناتج:

```
SAMMY SHARK
```

لنحوِّلها الآن إلى أحرفٍ صغيرة:

```
print(ss.lower())
```

وسينتج:

```
sammy shark
```

ستُسهِّل الدالتان `str.upper()` و `str.lower()` عملية التحقُّق من مساواة سلسلتين نصيتين لبعضهما أو لموازنتهما وذلك عبر توحيد حالة الأحرف. فلو كتب المستخدم اسمه بأحرف صغيرة فسنستطيع أن نتأكد إن كان مُسجَّلًا في قاعدة البيانات بموازنته بعد تحويل حالة أحرفه.

ب. الدوال المنطقية

تتوفر في بايثون عدّة دوال تتحقق من القيم المنطقية (Boolean). هذه الدوال مفيدةٌ عند

إنشائنا للنماذج التي يجب على المستخدمين ملأها؛ فمثلًا، إذا سألنا المستخدم عن الرمز البريدي وأردنا أن نقبل السلاسل النصية التي تحتوي أرقامًا فقط، أو عندما نسأله عن اسمه فسنقبل سلسلةً نصيةً تحوي حروفًا فقط. هنالك عددٌ من الدوال التي تُعيد قيمًا منطقيةً:

- `str.isalnum()`: تتحقَّق إذا احتوت السلسلة النصية على أرقام وأحرف فقط (دون رموز)، أي تعيد `true` إن تحقَّق ذلك.
- `str.isalpha()`: تتحقَّق إذا احتوت السلسلة النصية على أحرف فقط (دون أرقام أو رموز).
- `str.islower()`: تتحقَّق إذا كانت جميع أحرف السلسلة النصية صغيرة.
- `str.isnumeric()`: تتحقَّق إذا احتوت السلسلة النصية على أرقام فقط.
- `str.isspace()`: تتحقَّق إذا لم تحتوي السلسلة النصية إلا على الفراغات.
- `str.istitle()`: تتحقَّق إذا كانت حالة أحرف السلسلة النصية كما لو أنَّها عنوان باللغة الإنجليزية (أي أنَّ أوّل حرف من كل كلمة كبير، والبقية صغيرة).
- `str.isupper()`: تتحقَّق إذا كانت جميع أحرف السلسلة النصية كبيرة.

لنجرب استعمال بعضها عمليًا:

```
number = "5"
letters = "abcdef"

print(number.isnumeric())
print(letters.isnumeric())
```

الناتج:

```
True
False
```

استخدام الدالة `str.isnumeric()` على السلسلة النصية 5 سيُعيد القيمة `True`، بينما استخدام نفس الدالة على السلسلة النصية `abcdef` سيُعيد `False`. وبشكلٍ مماثل، يمكننا معرفة إن كانت حالة الأحرف في سلسلةٍ نصيةٍ كما لو أنَّها عنوان، أو أنها كبيرة أو صغيرة (هذه العملية تنطبق على اللغات المكتوبة بالأحرف اللاتينية). لنُنشِئ بدايةً بعض السلاسل النصية:

```
movie = "2001: A SAMMY ODYSSEY"
book = "A Thousand Splendid Sharks"
poem = "sammy lived in a pretty how town"
```

لنجرِّب الدوال المنطقيَّة لمعرفة الناتج (سنعرض كل دالتين وناتجها تحتهما):

```
print(movie.islower())
print(movie.isupper())

# False
# True

print(book.istitle())
print(book.isupper())

# True
# False

print(poem.istitle())
print(poem.islower())

# False
# True
```

ستساعدنا معرفة إن كانت أحرف السلسلة النصية بحالة صغيرة أو كبيرة أو كأنَّها عنوان في تصنيف البيانات تصنيفًا سليمًا، وتوفِّر لنا الفرصة لتوحيد طريقة تخزين البيانات بالتحقق من حالة أحرفها ثم تعديلها وفقًا لذلك. الدوال المنطقية التي تعمل على السلاسل النصية مفيدةٌ أيضًا عندما نريد التحقُّق إن حقَّقَت مدخلات المستخدم شروطًا معيَّنة.

ج. الدوال ‎join()‎ و ‎split()‎ و ‎replace()‎

توفِّر الدوال ‎`str.join()`‎ و ‎`str.split()`‎ و ‎`str.replace()`‎ إمكانياتٍ إضافيةً لتعديل السلاسل النصية في بايثون. الدالة ‎`str.join()`‎ تجمع سلسلتين نصيتين مع بعضهما، لكنَّها تفعل ذلك بتمرير إحداها إلى الأخرى. لنُنشِئ سلسلةً نصيةً:

```
balloon = "Sammy has a balloon."
```

لنستخدم الآن الدالة ‎`str.join()`‎ لإضافة فراغات إلى تلك السلسلة النصية كالآتي:

```
" ".join(balloon)
```

إذا طبعنا الناتج:

```
print(" ".join(balloon))
```

فسنجد أنَّ السلسلة النصية الجديدة هي السلسلة الأولى لكن بين كل حرفين فراغ:

```
S a m m y   h a s   a   b a l l o o n .
```

يمكننا أيضًا استخدام الدالة ‎`str.join()`‎ لإنشاء مقلوب سلسلة نصية:

```
print("".join(reversed(balloon)))
.noollab a sah ymmaS
```

لم نرغب في إضافة أيَّة سلسلة نصية إلى أخرى، لذا أبقينا على السلسلة النصية فارغةً دون

محتوى داخلها. الدالة `str.join()` مفيدةٌ أيضًا لجمع قائمة (list) من السلاسل النصية وإخراجها إلى سلسلةٍ وحيدة. لنُنشِئ سلسلة نصية يُفصَل بين كلماتها بفاصلة من القائمة الآتية:

```
print(",".join(["sharks", "crustaceans", "plankton"]))
sharks,crustaceans,plankton
```

إذا أردتَ وضع فاصلة ثم فراغ بين القيم في المثال السابق، فيمكنك أن تُعيد كتابة التعليمة البرمجية السابقة لإضافة فراغ بعد الفاصلة كما يلي:

```
", ".join(["sharks", "crustaceans", "plankton"])
```

وكما نستطيع جمع السلاسل النصية مع بعضها بعضًا، نستطيع أيضًا تجزئتها، وذلك عبر الدالة `str.split()`:

```
print(balloon.split())
['Sammy', 'has', 'a', 'balloon.']
```

ستُعيد الدالة `str.split()` قائمة (list) تحوي سلاسل نصية كانت مفصولةً بالفراغات في السلسلة النصية الأصلية إذا لم يُمرَّر معاملٌ لتحديد محرف الفصل. يمكنك أيضًا استخدام الدالة `str.split()` لحذف أجزاء معيَّنة من السلسلة النصية الأصلية، فلنحاول مثلًا حذف الحرف `a`:

```
print(balloon.split("a"))
['S', 'mmy h', 's ', ' b', 'lloon.']
```

حُذِفَ الحرف `a` من السلسلة النصية وأصبح الناتج مقسومًا عند كل ورود للحرف `a` مع الإبقاء على الفراغات. تأخذ الدالة `str.replace()` سلسلةً نصيةً وتُعيد نسخةً محدَّثةً منها بعد إجراء بعض عمليات الاستبدال عليها. لنفترض أنَّ البالون الذي يملكه سامي قد ضاع، ولعدم امتلاك سامي للبالون في الوقت الراهن، فسنُبدِّل الكلمة `"has"` إلى `"had"`:

```
print(balloon.replace("has","had"))
```

أوَّل سلسلة نصية داخل أقواس الدالة `replace()` هي السلسلة النصية التي نريد استبدالها، والسلسلة النصية الثانية هي السلسلة التي نريد وضعها بدلًا من الأولى. ناتج تنفيذ السطر السابق هو:

```
Sammy had a balloon.
```

استخدام دوال تعديل السلاسل النصية مثل `str.join()` و `str.split()` و `str.replace` سيمنحك تحكمًا كبيرًا بمعالجة السلاسل النصية في بايثون.

8. دوال الإحصاء

بعد أن تعرفنا على آلية فهرسة المحارف في السلاسل النصية، حان الوقت لمعاينة بعض الدوال التي تُحصي السلاسل النصية أو تعيد أرقام الفهارس. يمكننا أن نستفيد من ذلك بتحديد عدد المحارف التي نريد استقبالها من مدخلات المستخدم، أو لموازنة السلاسل النصية.

لدى السلاسل النصية -كغيرها من أنواع البيانات- عدّة دوال تُستخدم للإحصاء. لننظر أولًا إلى الدالة `len()` التي تُعيد طول أيِّ نوع متسلسل من البيانات، بما في ذلك الأنواع `string` و `list` و `tuple` و `dictionary`. لنطبع طول السلسلة النصية `ss`:

```
print(len(ss))
# 12
```

طول السلسلة النصية `"Sammy Shark!"` هو 12 محرفًا، بما في ذلك الفراغ وعلامة التعجب. بدلًا من استخدام متغيِّر، فلنحاول مباشرةً تمرير سلسلة نصية إلى الدالة `len()`:

```
print(len("Let's print the length of this string."))
# 38
```

الدالة `len()` تُحصي العدد الإجمالي من المحارف في سلسلة نصية. إذا أردنا إحصاء عدد مـرات تكـرار محـرف أو مجموعـة من المحـارف في سلسـلة نصـية، فيمكننـا اسـتخدام الدالـة `str.count()`، لنحاول إحصاء الحرف `a` في السلسلة النصية `ss`:

```
print(ss.count("a"))
# 2
```

يمكننا البحث عن محرفٍ آخر:

```
print(ss.count("s"))
# 0
```

صـحيحٌ أنَّ الحـرف `S` قد ورد في السلسـلة النصـية، إلا أنَّه من الضـروري أن تبقي بـذهنك أنَّ بايثون حساسةٌ لحالة الأحرف، فلو أردنا البحث عن حروفٍ معيَّنة بغض النظر عن حالتها، فعلينا حينها استخدام الدالة `str.lower()` لتحويل حروف السلسلة النصية إلى حروفٍ صغيرة.

لنحاول استخدام الدالة `str.count()` مع سلسلة من المحارف:

```
likes = "Sammy likes to swim in the ocean, likes to spin up
servers, and likes to smile."
print(likes.count("likes"))
```

النـاتج هـو 3، إذ تتواجـد مجموعـة المحـارف `"likes"` ثلاث مـرات في السلسـلة النصـية الأصلية. يمكننا أيضًا معرفة موقع الحرف أو مجموعة الحروف في السلسلة النصية، وذلك عبر الدالة `str.find()`، وسيُعاد موضع المحرف بناءً على رقم فهرسه. يمكننا أن نعرف متى يقع أوّل حرف `m` في السلسلة النصية `ss` كالآتي:

```
print(ss.find("m"))
# 2
```

أوّل مرة يقع فيها الحرف m في الفهرس 2 من السلسلة "Sammy Shark!"، يمكنك أن تراجع بداية هذا الدرس لرؤية جدول يبيّن ارتباطات المحارف مع فهارسها في السلسلة السابقة.

لنرى الآن مكان أوّل ظهور لمجموعة المحارف likes في السلسلة "likes":

```
print(likes.find("likes"))
# 6
```

أوّل مرة تظهر فيها السلسلة "likes" هي في الفهرس 6، أي مكان وجود الحرف l من likes. ماذا لو أردنا أن نعرف موضع ثاني تكرار للكلمة likes؟ نستطيع فعل ذلك بتمرير معاملٍ ثانٍ إلى الدالة str.find() الذي سيجعلها تبدأ بحثها من ذاك الفهرس، فبدلًا من البحث من أوّل السلسلة النصية سنبحث انطلاقًا من الفهرس 9:

```
print(likes.find("likes", 9))
# 34
```

بدأ البحث في هذا المثال من الفهرس 9، وكانت أوّل مطابقة للسلسلة النصية "likes" عند الفهرس 34. إضافةً إلى ذلك، يمكننا تحديد نهاية إلى مجال البحث بتمرير معامل ثالث. وكما عند تقسيم السلاسل النصية، يمكننا استخدام أرقام الفهارس السالبة للعد عكسيًا:

```
print(likes.find("likes", 40, -6))
# 64
```

يبحث آخر مثال عن موضع السلسلة النصية "likes" بين الفهرس 40 و -6، ولمّا كان المعامل الأخير هو رقم سالب، فسيبدأ العد من نهاية السلسلة الأصلية.

دوال الإحصاء مثل `len()` و `str.count()` و `str.find()` مفيدةٌ في تحديد طول السلسلة النصية وعدد حروفها وفهارس ورود محارف معيَّنة فيها.

9. خلاصة الفصل

لقد تعلمنا في هذا الفصل أساسيات التعامل مع السلاسل النصية في لغة بايثون 3. بما في ذلك إنشاءها وطباعتها وجمعها وتكرارها، إضافةً إلى تخزينها في متغيرات، وهذه هي المعلومات الأساسية التي عليك فهمها للانطلاق في تعاملك مع السلاسل النصية في برامج بايثون 3.

8

مدخل إلى تنسيق النصوص

تتألف السلاسل النصية عادةً من النص المكتوب، وهنالك عدّة حالات نحتاج فيها إلى تحكمٍ أكبر بكيفيَّة إظهار النص وجعلها أسهل قراءةً للبشر عبر وضع علامات الترقيم والسطور الجديدة والمحاذاة. سنشرح في هذا الفصل كيفيَّة التعامل مع السلاسل النصية في بايثون لكي يظهر النص الناتج بتنسيق صحيح.

1. الصياغة المختزلة

لنفرِّق أولًا بين الصياغة المختزلة للسلاسل النصية (string literal) والسلاسل النصية المجرَّدة نفسها (string value)، فالأولى هي ما نراه في الشيفرة المصدرية للبرنامج، بما في ذلك علامتَي الاقتباس. أمَّا السلسلة النصية نفسها فهي ما نراها عندما نستدعي الدالة `print()` عند تشغيل البرنامج. ففي برنامج Hello, World! التقليدي، تكون الصياغة المختزلة هي `"Hello, World!"` بينما السلسلة النصية المجرَّدة هي `Hello, World!` دون علامتَي الاقتباس. أي أنَّ السلسلة النصية هي ما نراه في نافذة الطرفية عندما نُشغِّل برنامج بايثون. لكن بعض السلاسل النصية قد تحتوي على علامات اقتباس، مثل اقتباسنا لمقولةٍ ما. ولأنَّ القيم المُصنَّفة على أنَّها سلاسل نصية بالصياغة المختزلة والقيم الفعلية المجرَّدة للسلاسل النصية غير متساوية، فمن الضروري في أغلب الحالات إضافة تنسيق إلى صياغة السلسلة النصية المختزلة لعرضها كما ينبغي.

2. علامات الاقتباس

بسبب إمكانيتنا استخدام علامات الاقتباس المفردة أو المزدوجة في بايثون، فمن السهل تضمين الاقتباسات بوضعها بين علامتَي اقتباس مزدوجتين في سلسلةٍ نصيةٍ محاطةٍ بعلامتَي اقتباس مفردتين كما في السلسلة الآتية:

```
'Sammy says, "Hello!"'
```

أو يمكننا استخدام علامة اقتباس فردية (أو كما يسمونها «فاصلة عليا» [apostrophe]) في سلسلةٍ نصيةٍ محاطةٍ بعلامتَي اقتباس مزدوجتين:

```
"Sammy's balloon is red."
```

إذًا، يمكننا التحكم بطريقة عرض علامات الاقتباس والفواصل العليا في سلاسلنا النصية عبر استخدام النوع الصحيح من علامات الاقتباس لإحاطة كامل السلسلة النصية.

3. كتابة النص على أكثر من سطر

طباعة السلاسل النصية على أكثر من سطر ستجعل منها واضحةً وسهلة القراءة. إذ يمكن تجميع النصوص المكتوبة بعدّة أسطر لزيادة وضوحها، أو لتنسيقها كرسالة، أو للحفاظ على تعدُّد الأسطر في الأشعار. نستخدم ثلاث علامات اقتباس فردية `'''` أو ثلاث علامات اقتباس مزدوجة `"""` للإحاطة بالسلسلة النصية التي تمتد على أكثر من سطر:

```
'''
This string is on
multiple lines
within three single
quotes on either side.
'''
"""
This string is on
multiple lines
within three double
quotes on either side.
"""
```

يمكنك الآن طباعة السلاسل النصية في عدّة أسطر لجعل النصوص سهلة القراءة -خصوصًا الطويلة منها- وذلك عبر استخدام ثلاث علامات اقتباس متتالية.

4. تهريب المحارف

طريقة أخرى لتنسيق السلاسل النصية هي استخدام «محرف التهريب» (escape character). فجميع عمليات تهريب المحارف تبدأ بالخط المائل الخلفي (backslash، أي \) متبوعًا بمحرفٍ آخر الذي له معنى خاص يفيد في تنسيق السلسلة النصية. هذه قائمة بأكثر محارف التهريب شيوعًا:

- `\`: سطرٌ جديدٌ في سلسلةٍ نصيةٍ متألفةٍ من عدّة أسطر.
- `\\`: طباعة رمز الخط المائل الخلفي.
- `\'`: طباعة علامة اقتباس فردية.
- `\"`: طباعة علامة اقتباس مزدوجة.
- `\n`: طباعة محرف الانتقال إلى سطرٍ جديد.
- `\t`: طباعة محرف الجدولة (Tab).

لنستخدم محرف التهريب لإضافة علامات الاقتباس إلى سلسلتنا النصية السابقة، لكن هذه المرة سنُحيط السلسلة النصية بعلامتَي اقتباس مزدوجتين:

```
print("Sammy says, \"Hello!\"")
# Sammy says, "Hello!"
```

تمكننا عبر محرف التهريب "\ من استخدام علامات الاقتباس المزدوجة للإحاطة بالسلسلة النصية التي تحتوي على نصٍ مقتبسٍ ومحاطٍ بعلامتَي اقتباس مزدوجتين. نستطيع أيضًا استخدام محرف التهريب '\ لإضافة علامة اقتباس مفردة ضمن السلسلة النصية المحاطة بعلامتَي اقتباس مفردتين:

```
print('Sammy\'s balloon is red.')
# Sammy's balloon is red.
```

ولأننا نستخدم الآن محرف التهريب فنستطيع وضع علامات الاقتباس المفردة حتى لو كانت السلسلة النصية كلها موجودة بين علامتَي اقتباس مفردتين. عندما نستخدم علامات الاقتباس الثلاثية -كما فعلنا أعلاه- فسنجد فراغًا في أعلى وأسفل النص عند طباعته. نستطيع حذف تلك الفراغات عبر استخدام محرف التهريب \ في بداية ونهاية السلسلة النصية مع الإبقاء على النص مقروءًا بسهولة في الشيفرة.

```
"""\
This multi-line string
has no space at the
top or the bottom
when it prints.\
"""
```

وبشكلٍ شبيهٍ بما سبق، يمكننا استخدام محرف التهريب n\ لوضع أسطر جديدة دون الحاجة إلى الضغط على زر Enter أو Return:

```
print("This string\nspans multiple\nlines.")
# This string
# spans multiple
# lines.
```

يمكننا الدمج بين محارف التهريب، إذ سنطبع في المثال الآتي سلسلةً نصيةً على أكثر من سطر، ونستعمل فيها مسافة جدولة (tab) بين الترقيم ومحتوى السطر:

```
print("1.\tShark\n2.\tShrimp\n10.\tSquid")
# 1.    Shark
# 2.    Shrimp
# 10.    Squid
```

علامة الجدولة الأفقية التي وضعناها عبر محرف التهريب `\t` ستحاذي الكتابة في العمود النصي الثاني في المثال أعلاه، مما يجعل قراءتها سهلةً جدًا. وصحيحٌ أنَّ محرف التهريب `\n` يعمل عملًا جيدًا في النصوص القصير، لكن لا نُغفِل أهمية أن تكون الشيفرة المصدرية مقروءةً بسهولةٍ أيضًا. فلو كان النص طويلًا، فأرى أنَّ من الأفضل استخدام علامات الاقتباس الثلاثية.

رأينا أنَّ محارف التهريب تُستعمَل لإضافة تنسيق إلى السلاسل التي كان من الصعب (أو حتى المستحيل) عرضها عرضًا سليمًا دونها. فهل تستطيع مثلًا أن تطبع السلسلة النصية الآتية دون استخدام محارف التهريب؟

```
Sammy says, "The balloon's color is red."
```

5. السلاسل النصية الخام

ماذا لو أردنا تجاهل كل محارف التنسيق الخاصة في سلاسلنا النصية؟ فلربما أردنا موازنة أو التحقُّق من صحة بعض الشيفرات الحاسوبية التي تستخدم الخط المائل الخلفي، ولا نريد من

بايثون تفسيره على أنَّه محرف تهريب. أتت «السلاسل النصية الخام» (raw strings) في بايثون لتحل هذه المشكلة، وتتجاهل جميع محارف التنسيق داخل سلسلة نصية، بما في ذلك محارف التهريب.

يمكننا إنشاء سلسلة نصية خام بوضع الحرف r في بداية السلسلة النصية، قبل علامة الاقتباس الأولى مباشرةً:

```
print(r"Sammy says,\"The balloon\'s color is red.\"")
# Sammy says,\"The balloon\'s color is red.\"
```

سنستطيع الإبقاء على محارف التهريب كما هي في السلاسل النصية إن أسبقناها بالحرف r لتحويلها إلى سلاسل نصية خام.

6. استخدام المُنسِّقات

الدالة str.format() المتوافرة للسلاسل النصية تسمح لك باستبدال المتغيرات وتنسيق القيم. مما يمنحك القدرة على تجميع العناصر مع بعضها عبر إدخالها في مواضع معينة. سيشرح لك هذا القسم أشهر الاستخدامات لآلية تنسيق السلاسل النصية في بايثون، والتي ستساعدك في جعل شيفرتك وبرنامجك أسهل قراءةً واستخدامًا.

تعمل المُنسِّقات (formatters) بوضع حقول قابلة للاستبدال تُعرَّف عبر وضع قوسين معقوفين {} في السلسلة النصية ثم استدعاء الدالة str.format()، إذ ستُمرَّر القيمة التي تريد وضعها ضمن السلسلة النصية إلى الدالة format() وستوضع هذه القيمة في نفس مكان الحقل القابل للاستبدال الموجود في السلسلة الأصلية عندما تُشغِّل برنامجك. لنطبع سلسلةً نصيةً تستخدم «منسِّقًا» (formatter):

```
print("Sammy has {} balloons.".format(5))
```

الناتج:

```
Sammy has 5 balloons.
```

أنشأنا في المثال السابق سلسلةً نصيةً تحتوي على قوسين معقوصين `{}`:

```
"Sammy has {} balloons."
```

ثم أضفنا الدالة `str.format()` ومرَّرنا إليها القيمة الرقمية 5 وهذا يعني أنَّ القيمة 5 ستوضع مكان القوسين المعقوصين:

```
Sammy has 5 balloons.
```

يمكننا أيضًا إسناد السلسلة النصية الأصلية التي تحوي مُنسِّقًا إلى متغير:

```
open_string = "Sammy loves {}."
print(open_string.format("open source"))
```

الناتج:

```
Sammy loves open source.
```

أضفنا في المثال السابق السلسلة النصية `"open source"` إلى سلسلةٍ نصيةٍ أكبر باستبدالها للقوسين المعقوفين الموجودَين في السلسلة الأصلية. تسمح لك المُنسِّقات في بايثون باستخدام الأقواس المعقوفة لحجز أماكن للقيم التي ستمررها مستقبلًا عبر الدالة `str.format()`.

ا. استخدام المُنسِّقات لحجز أكثر من مكان

يمكنك استخدام أكثر من زوج من الأقواس المعقوصة عند استعمال المُنسِّقات؛ فيمكنك أن

تضيف سلسلةً نصيةً أخرى إلى المثال السابق وذلك بإضافة زوج آخر من الأقواس المعقوصة وتمرير قيمة ثانية إلى الدالة كما يلي:

```
# {} مكانين محجوزين عبر
new_open_string = "Sammy loves {} {}."
# تمرير قيمتين إلى الدالة مفصولٌ بينهما بفاصلة
print(new_open_string.format("open-source", "software"))
```

الناتج:

```
Sammy loves open-source software.
```

أضفنا زوجًا آخر من الأقواس المعقوصة إلى السلسلة النصية للسماح بوضع قيمة ثانية، ثم مررنا سلسلتين نصيتين إلى الدالة `str.format()` مفصولٌ بينهما بفاصلة. سنضيف عمليات استبدال أخرى عبر اتباع نفس الآلية التي شرحناها أعلاه:

```
sammy_string = "Sammy loves {} {}, and has {} {}."
print(sammy_string.format("open-source", "software", 5,
"balloons"))
```

الناتج:

```
Sammy loves open-source software, and has 5 balloons.
```

ب. إعادة ترتيب المنسقات عبر المعاملات الموضعية

عندما نترك الأقواس المعقوصة دون معاملات (parameters) ممررة إليها، فستضع بايثون القيم المُمرَّرة إلى الدالة `str.format()` بالترتيب. هذا تعبيرٌ فيه زوجين من الأقواس المعقوصة يوضع مكانهما سلسلتان نصيتان شبيهةٌ بما رأيناه سابقًا في هذا الفصل:

```
print("Sammy the {} has a pet {}!".format("shark", "pilot
fish"))
```

الناتج:

```
Sammy the shark has a pet pilot fish!
```

اُستبدِل أوّل زوجٍ من الأقواس المعقوصة ووضعت مكانه القيمة `"shark"`، ووضعت القيمة `"pilot fish"` مكان الزوج الثاني من الأقواس. القيم التي مرَّرناها إلى الدالة `str.format()` كانت بهذا الترتيب:

```
("shark", "pilot fish")
```

لاحظ أنَّ القيمة السابقة هي من النوع `tuple` (صف)، ويمكن الوصول إلى كل قيمة موجودة فيها عبر فهرسٍ رقمي تابعٍ لها، والذي يبدأ من الفهرس 0. يمكننا تمرير أرقام الفهارس إلى داخل القوسين المعقوفين:

```
print("Sammy the {0} has a pet {1}!".format("shark", "pilot
fish"))
```

سنحصل بعد تنفيذ المثال السابق على نفس الناتج التي ظهر دون تحديد أرقام الفهارس يدويًّا، وذلك لأنَّنا استدعينا القيم بالترتيب:

```
Sammy the shark has a pet pilot fish!
```

لكن إن عكسنا أرقام الفهارس في معاملات الأقواس المعقوفة فسنتمكن من عكس ترتيب القيم المُمرَّرة إلى السلسلة النصية الأصلية:

```
print("Sammy the {1} has a pet {0}!".format("shark", "pilot
fish"))
```

الناتج:

```
Sammy the pilot fish has a pet shark!
```

لكن إن حاولت استخدام الفهرس ذي الرقم 2 ولم تكن لديك إلا قيمتين موجودتين في الفهرسين 0 و 1، فأنت تستدعي قيمةً خارج المجال المسموح، ولهذا السبب ستظهر رسالة خطأ:

```
print("Sammy the {2} has a pet {1}!".format("shark", "pilot fish"))
```

الناتج:

```
IndexError: tuple index out of range
```

تُشير رسالة الخطأ إلى وجود قيمتين فقط ومكانهما هو 0 و1، لذا كان الفهرس 2 غير مرتبطٍ بقيمةٍ وكان خارج المجال المسموح. لنضف الآن مكانين محجوزين إلى السلسلة النصية ولنمرِّر بضع قيم إلى الدالة `str.format()` لكي نفهم آلية إعادة الترتيب فهمًا تامًا. هذه هي السلسلة النصية الجديدة التي فيها أربعة أزواج من الأقواس المعقوصة:

```
print("Sammy is a {}, {}, and {} {}!".format("happy", "smiling", "blue", "shark"))
```

الناتج:

```
Sammy is a happy, smiling and blue shark!
```

ستوضع القيم المُمرَّرة إلى الدالة `str.format()` بنفس ترتيب ورودها في حال لم نستعمل المعاملات داخل الأقواس المعقوصة. تملك السلاسل النصية المُمرَّرة إلى الدالة `str.format()` الفهارس الآتية المرتبطة بها؛ لنستخدم الآن أرقام الفهارس لتغيير ترتيب ظهور القيم المرتبطة بها في السلسلة النصية:

```
print("Sammy is a {3}, {2}, and {1} {0}!".format("happy",
"smiling", "blue", "shark"))
```

الناتج:

```
Sammy is a shark, blue, and smiling happy!
```

ولمّا كنّا قد بدأنا بالفهرس ذي الرقم 3، فستظهر القيمة "shark" أولًا أي أنَّ وضع رقم الفهرس بين القوسين كمعامل سيؤدي إلى تغيير ترتيب ظهور القيم في السلسلة النصية الأصلية.

نستطيع -بالإضافة إلى المعاملات الموضعية الرقمية- أن نربط بين القيم وبين كلمات محجوزة مخصصة ومن ثم نستدعيها عبر وضع الكلمة المحجوزة بين القوسين المعقوصين كما يلي:

```
print("Sammy the {0} {1} a {pr}.".format("shark", "made", pr =
"pull request"))
```

الناتج:

```
Sammy the shark made a pull request.
```

أظهر المثال السابق استخدام كلمة محجوزة وسيطًا بالإضافة إلى المعاملات الموضعيَّة، يمكنك استخدام الكلمة المحجوزة pr كوسيط بالإضافة إلى أرقام الفهارس، وتستطيع أيضًا إعادة ترتيب تلك الوسائط كيفما شئت:

```
print("Sammy the {pr} {1} a {0}.".format("shark", "made", pr =
"pull request"))
```

الناتج:

```
Sammy the pull request made a shark.
```

استخدام المعاملات الموضعيَّة والكلمات المحجوزة سيمنحنا تحكمًا أكبر بكيفيَّة معالجة السلسلة النصية الأصلية عبر إعادة ترتيب القيم المُمرَّرة إليها.

ج. استخدام المُنسِّقات لتنظيم البيانات

يسطع نجم آلية التنسيق التي نشرحها في هذا الفصل عندما تُستخدَم لتنظيم البيانات بصريًا، فلو أردنا إظهار نتائج قاعدة البيانات إلى المستخدمين، فيمكننا استعمال المُنسِّقات لزيادة حجم الحقل وتعديل المحاذاة لجعل الناتج أسهل قراءةً. لننظر إلى حلقة تكرار تقليدية في بايثون التي تطبع `i` و `i*i` و `i*i*i` لمجالٍ من الأعداد من 3 إلى 13:

```
for i in range(3,13):
   print(i, i*i, i*i*i)
```

الناتج:

```
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
11 121 1331
12 144 1728
```

صحيحٌ أنَّ الناتج مُنظَّمٌ قليلًا، إلا أنَّ الأعداد تتداخل مع بعضها بصريًا مما يُصعِّب قراءة الأسطر الأخيرة من الناتج، وإذا كنتَ تتعامل مع مجموعة أكبر من البيانات التي يتواجد فيها أعداد أكبر (أو أصغر) مما عرضناه في مثالنا، فقد تبدو لك المشكلة جليةً حينها. لنحاول تنسيق

الناتج السابق لإعطاء مساحة أكبر لإظهار الأعداد عبر المثال التالي:

```
for i in range(3,13):
    print("{:3d} {:4d} {:5d}".format(i, i*i, i*i*i))
```

لم نُحدّد في المثال السابق ترتيب الحقل وبدأنا مباشرةً بكتابة النقطتين الرأسيتين متبوعةً بحجم الحقل ورمز التحويل d (لأنَّنا نتعامل مع أعداد صحيحة). أعطينا في المثال السابق حجمًا للحقل مساويًا لعدد أرقام العدد الذي نتوقع طباعته في الحقل المعني مضافًا إليه 2، لذا سيبدو الناتج كالآتي:

```
  3   9    27
  4  16    64
  5  25   125
  6  36   216
  7  49   343
  8  64   512
  9  81   729
 10 100  1000
 11 121  1331
 12 144  1728
```

يمكننا أيضًا تحديد حجم ثابت للحقل لنحصل على أعمدة متساوية العرض، مما يضمن إظهار الأعداد الكبيرة بصورة صحيحة:

```
for i in range(3,13):
    print("{:6d} {:6d} {:6d}".format(i, i*i, i*i*i))
```

الناتج:

```
     3      9     27
     4     16     64
```

```
5    25    125
6    36    216
7    49    343
8    64    512
9    81    729
10    100   1000
11    121   1331
12    144   1728
```

يمكننا أيضًا تعديل محاذاة النص الموجود في الأعمدة باستخدام الرموز < و ^ و >، وتبديل d إلى f لإظهار منازل عشرية، وغير ذلك مما تعلمناه في هذا الدرس لإظهار البيانات الناتجة كما نرغب.

7. تحديد نوع القيمة

يمكنك وضع معاملات أخرى ضمن القوسين المعقوصين، سنستخدم الصيغة الآتية `{field_name:conversion}` إذ `field_name` هو الفهرس الرقمي للوسيط المُمرَّر إلى الدالة `str.format()` والذي شرحناه تفصيليًا في القسم السابق، و `conversion` هو الرمز المستعمل للتحويل إلى نوع البيانات الذي تريده. «رمز التحويل» يعني رمزًا من حرفٍ وحيد الذي تستخدمه بايثون لمعرفة نوع القيمة المُراد «تنسيقها». الرموز التي سنستخدمها في أمثلتنا هي `s` للسلاسل النصية و `d` لإظهار الأرقام بنظام العد العشري (ذي الأساس 10) و `f` لإظهار الأعداد ذات الفاصلة.

يمكنك قراءة المزيد من التفاصيل عن رموز التنسيق في بايثون 3 (وغير ذلك من المواضيع المرتبطة بهذا المجال) في التوثيق الرسمي. لننظر إلى مثالٍ نُمرِّر فيه رقمًا صحيحًا عبر الدالة `format()` لكننا نريد إظهاره كعددٍ ذي فاصلة عبر رمز التحويل `f`:

```
print("Sammy ate {0:f} percent of a {1}!".format(75, "pizza"))
```

الناتج:

```
Sammy ate 75.000000 percent of a pizza!
```

وضعت القيمة -مكان أوّل ورود للصيغة `{field_name:conversion}`- كعددٍ ذي فاصلة، أمّا ثاني ورود للقوسين المعقوصين فكان بالصيغة `{field_name}`. لاحظ في المثال السابق وجود عدد كبير من الأرقام الظاهرة بعد الفاصلة العشرية، لكنك تستطيع تقليل عددها. فعندما نستخدم الرمز `f` للقيم ذات الفاصلة نستطيع أيضًا تحديد دقة القيمة الناتجة بتضمين رمز النقطة `.` متبوعًا بعدد الأرقام بعد الفاصلة التي نود عرضها. حتى لو أكل سامي 75،765367% من قطعة البيتزا فلن نحتاج إلى هذا القدر الكبير من الدقة، إذ يمكننا مثلًا أن نجعل عدد المنازل العشرية ثلاث منازل بعد الفاصلة بوضعنا `.3` قبل رمز التحويل `f`:

```
print("Sammy ate {0:.3f} percent of a
pizza!".format(75.765367))
```

الناتج:

```
Sammy ate 75.765 percent of a pizza!
```

أما إذا أردنا عرض منزلة عشرية وحيدة، فيمكننا إعادة كتابة السلسلة السابقة كالآتي:

```
print("Sammy ate {0:.1f} percent of a
pizza!".format(75.765367))
```

الناتج:

```
Sammy ate 75.8 percent of a pizza!
```

لاحظ كيف أدى تعديل دقة الأرقام العشرية إلى تقريب الرقم (وفق قواعد التقريب الاعتيادية). وصحيحٌ أننا عرضنا رقمًا دون منازل عشرية كعددٍ ذي فاصلة، إلا أننا إذا حاولنا تحويل عدد عشري إلى عدد صحيح باستخدام رمز التحويل d فسنحصل على خطأ:

```
print("Sammy ate {0:.d} percent of a pizza!".format(75.765367))
```

الناتج:

```
ValueError: Unknown format code 'd' for object of type 'float'
```

إذا لم ترغب بعرض أيّة منازل عشرية، فيمكنك كتابة تعبير كالآتي:

```
print("Sammy ate {0:.0f} percent of a
pizza!".format(75.765367))
```

الناتج:

```
Sammy ate 76 percent of a pizza!
```

لن يؤدّي ما سبق إلى تحويل العدد العشري إلى عددٍ صحيح، وإنّما سيؤدي إلى تقليل عدد المنازل العشرية الظاهرة بعد الفاصلة.

8. إضافة حواشي

لمّا كانت الأماكن المحجوزة عبر القوسين المعقوصين {} هي حقول قابلة للاستبدال (أي ليست قيمًا فعليةً) فيمكنك إضافة حاشية (padding) أو إضافة فراغ حول العنصر بزيادة حجم الحقل عبر معاملات إضافية، قد تستفيد من هذا الأمر عندما تحتاج إلى تنظيم البيانات بصريًا.

يمكننا إضافة حقلٍ بحجمٍ معيّن (مُقاسًا بعدد المحارف) بتحديد ذاك الحجم بعد النقطتين الرأسيتين : كما في المثال الآتي:

```
print("Sammy has {0:4} red {1:16}!".format(5, "balloons"))
```

الناتج:

```
Sammy has    5 red balloons        !
```

أعطينا في المثال السابق حقلًا بحجم 4 محارف للعدد 5، وأعطينا حقلًا بحجم 16 محرفًا للسلسلة النصية balloons (لأنها سلسلة طويلة نسبيًا). وكما رأينا من ناتج المثال السابق، يتم محاذاة السلاسل النصية افتراضيًا إلى اليسار والأعداد إلى اليمين، يمكنك أن تُغيِّر من هذا بوضع رمز خاص للمحاذاة بعد النقطتين الرأسيتين مباشرةً. إذ سيؤدي الرمز < إلى محاذاة النص إلى يسار الحقل، أما الرمز ^ فسيوسِّط النص في الحقل، والرمز > سيؤدي إلى محاذاته إلى اليمين.

لنجعل محاذاة العدد إلى اليسار ونوسِّط السلسلة النصية:

```
print("Sammy has {0:<4} red {1:^16}!".format(5, "balloons"))
```

الناتج:

```
Sammy has 5    red     balloons    !
```

نلاحظ الآن أنَّ محاذاة العدد 5 إلى اليسار، مما يعطي مساحة فارغةً في الحقل قبل الكلمة red، وستظهر السلسلة النصية balloons في منتصف الحقل وتوجد مسافة فارغة على يمينها ويسارها. عندما نُنسِّق الحقل لنجعله أكبر من حجمه الطبيعي فستملأ بايثون الحقل افتراضيًا بالفراغات، إلا أنَّنا نستطيع تغيير محرف الملء إلى محرفٍ آخر بوضعه مباشرةً بعد النقطتين الرأسيتين:

```
print("{:*^20s}".format("Sammy"))
```

الناتج:

```
*******Sammy********
```

ستضع بايثون السلسلة النصية المُمرَّرة إلى الدالة `str.format()` ذات الفهرس 0 مكان القوسين المعقوصين لأنَّنا لم نطلب منها عكس ذلك، ومن ثم سنضع النقطتين الرأسيتين ثم سنُحدِّد أننا سنستعمل المحرف * بدلًا من الفراغات لملء الحقل، ثم سنوسِّط السلسلة النصية عبر استعمال الرمز ^ مُحدِّدين أنَّ حجم الحقل هو 20 محرف، ومُشيرين في نفس الوقت إلى أنَّ الحقل هو حقلٌ نصيٌ عبر وضع الرمز `s`. يمكننا استخدام هذه المعاملات مع المعاملات التي استخدمناها وشرحناها في الأقسام السابقة:

```
print("Sammy ate {0:5.0f} percent of a pizza!".format(75.765367))
```

الناتج:

```
Sammy ate    76 percent of a pizza!
```

حدَّدنا داخل القوسين المعقوصين رقم فهرس القيمة العددية ثم وضعنا النقطتين الرأسيتين ثم اخترنا حجم الحقل ثم وضعنا نقطةً . لنضبط عدد المنازل العشرية الظاهرة، ثم نختار نوع الحقل عبر رمز التحويل `f`.

9. استخدام المتغيرات

مرَّرنا منذ بداية هذا الفصل وإلى الآن الأعداد والسلاسل النصية إلى الدالة `str.format()` مباشرةً، لكنَّنا نستطيع تمرير المتغيرات أيضًا، وذلك بنفس الآلية المعتادة:

```
nBalloons = 8
print("Sammy has {} balloons today!".format(nBalloons))
```

الناتج:

```
Sammy has 8 balloons today!
```

يمكننا أيضًا استخدام المتغيّرات لتخزين السلسلة النصية الأصلية بالإضافة إلى القيم التي ستُمرَّر إلى الدالة:

```
sammy = "Sammy has {} balloons today!"
nBalloons = 8
print(sammy.format(nBalloons))
```

الناتج:

```
Sammy has 8 balloons today!
```

ستُسهِّل المتغيرات من التعامل مع تعبيرات التنسيق وتُبسِّط عملية إسناد مدخلات المستخدم وإظهارها مُنسَّقةً في السلسلة النصية النهائية.

10. خلاصة الفصل

شرحنا في هذا الفصل عدّة طرائق لتنسيق النصوص والسلاسل النصية في بايثون 3؛ وعرفنا كيف نُظهِر السلاسل النصية كما نريد عبر استخدامنا لمحارف التهريب أو السلاسل النصية الخام والمُنسِّقات، لكي يستفيد المستخدم من النص الناتج ويقرأه بسهولة.

9

العمليات الحسابية

الأعداد شائعةٌ جدًا في البرمجة، إذ تُستخدم لتمثيل مختلف القيم، مثل أبعاد حجم الشاشة، والمواقع الجغرافية، والمبالغ المالية، ومقدار الوقت الذي مر منذ بداية فيديو، والألوان وغير ذلك.

تعد القدرة على تنفيذ العمليات الرياضية بفعالية في البرمجة مهارة مهمة، لأنَّك الأعداد ستكون في متناول الأيدي دومًا. الفهم الجيد للرياضيات يمكن أن يساعدك على أن تصبح مبرمجًا أفضل، إلا أنَّه ليس شرطًا أساسيًا. فالرياضيات أداة لتحقيق ما ترغب في تحقيقه، وطريقة لتحسين آلية التنفيذ.

سنعمل مع أكثر نوعي البيانات استخدامًا في بايثون، وهما الأعداد الصحيحة والأعداد العشرية:

- الأعداد الصحيحة هي أعداد كاملة يمكن أن تكون موجبة أو سالبة أو معدومة مثل (...، -1 ، 0 ، 1، ...).
- الأعداد العشرية هي أعداد حقيقية تحتوي على فاصلة عشرية (كما في `9.0` أو `-2.25`).

سنلقي في هذا الفصل نظرةً على العوامل (operators) التي يمكن استخدامها مع أنواع البيانات العددية في بايثون.

1. العاملات

العامل (operator) هو رمز أو دالة تمثل عمليَّة حسابية، إذ جاءت تسمية عامل من عملية، أي العامل الذي يجري عمليَّة. على سبيل المثال، في الرياضيات، علامة الجمع أو + هي العامل الذي يجري عملية الجمع.

في بـايثون، سـنرى بعض العـاملات المألوفـة، والـتي اسـتُعيرَت من الرياضـيات، لكن هنـاك عاملات أخرى خاصة بمجال البرمجة.

الجدول التالي مرجعٌ سريعٌ للعاملات الحسابية في بايثون. سنغطي جميع هذه العمليات في هذا الفصل.

العملية	الناتج
x + y	مجموع x و y
x - y	طرحx من y
-x	عكس إشارة x
+x	نفس قيمة x
x * y	ضربx بـ y
x / y	قسمة x على y
x // y	حاصل القسمة التحتية لـ x على y
x % y	باقي قسمة x على y
x ** y	x أس y

سـنتحدث أيضًـا عن عـاملات الإسـناد المركبـة (compound assignment operators)، بما في ذلك =+ و =*، التي تجمع عاملًا حسابيًا مع العامل =.

2. الجمع والطرح

في بايثون، يعمل معاملا الجمع والطرح مثل ما هو معروف في الرياضيات. في الواقع، يمكنك استخدام لغة بايثون آلةً حاسبةً.

لنلقِ نظرة على بعض الأمثلة، بدءًا من الأعداد الصحيحة:

```
print(1 + 5)
```

والناتج:

```
6
```

بدلًا من تمرير أعداد صحيحة مباشرة إلى الدالة `print`، يمكننا تهيئة المتغيرات بأعداد صحيحة:

```
a = 88
b = 103

print(a + b)
```

وسينتج لنا:

```
191
```

الأعداد الصحيحة يمكن أن تكون موجبة أو سالبة (أو معدومة أيضًا)، لذلك يمكننا إضافة عدد سالب إلى عدد موجب:

```
c = -36
d = 25

print(c + d)  #  -11
```

الجمع سيكون مشابهًا مع الأعداد العشرية:

```
e = 5.5
f = 2.5

print(e + f)  #  8.0
```

إذا جمعنا عددين عشريين معًا، ستعيد بايثون عددًا عشريًّا.

صياغة الطرح تشبه صياغة الجمع، ما عدا أنك ستستبدل بعامل الطرح (-) عامل الجمع (+):

```
g = 75.67
h = 32

print(g - h)   #  43.67
```

هنا، طرحنا عددًا صحيحًا من عدد عشري. ستعيد بايثون عددًا عشريًّا إذا كان أحد الأعداد المتضمنة في المعادلة عشريًّا.

3. العمليات الحسابية الأحادية

يتكون التعبير الرياضي الأحادي (unary mathematical expression) من مكوّن أو عنصر واحد فقط، ويمكن في بايثون استخدام العلامتين + و - بمفردهما عبر قرنهما بقيمة لإعادة القيمة نفسها (عبر العامل +)، أو تغيير إشارة القيمة (عبر العامل -) .

تشير علامة الجمع - رغم أنَّها لا تُستخدم كثيرًا - إلى هوية القيمة (identity of the value)، أي تعيد القيمة نفسها. يمكننا استخدام علامة الجمع مع القيم الموجبة:

```
i = 3.3
print(+i)  #  3.3
```

عندما نستخدم علامة الجمع مع قيمة سالبة، فستُعاد القيمة نفسها، وفي هذه الحالة ستكون قيمة سالبة أيضًا:

```
j = -19
print(+j)    #  -19
```

علامة الطرح، على خلاف علامة الجمع، تغيِّر إشارة القيمة. لذلك، عندما نضعها مع قيمة موجبة، ستُعاد القيمة السالبة منها:

```
i = 3.3
print(-i)   #  -3.3
```

بالمقابل، عندما نستخدم عامل الطرح الأحادي (minus sign unary operator) مع قيمة سالبة، فستُعاد القيمة الموجبة منها:

```
j = -19
print(-j)   # 19
```

ستُعيِد العمليتان الحسابيتان الأحاديتان + و - إمَّا هوية القيمة المعطاة، أو القيمة المعاكسة في الإشارة للقيمة المعطاة على التوالي.

4. الضرب والقسمة

مثل الجمع والطرح، الضرب والقسمة في بايثون مشابهان لما هو معروف في الرياضيات. علامة الضرب في بايثون هي *، وعلامة القسمة هي /.

فيما يلي مثال على ضرب عددين عشريين في بايثون:

```
k = 100.1
```

```
l = 10.1

print(k * l)    #  1011.0099999999999
```

عندما تُجرَى عملية القسمة في بايثون 3، فسيكون العدد المُعاد دائمًا عشريًّا، حتى لو استخدمت عددين صحيحين:

```
m = 80
n = 5

print(m / n)    #  16.0
```

هذا أحد الاختلافات الرئيسية بين بايثون 2 و بايثون 3. الإجابة في بايثون 3 تكون كسرية، فعند استخدام / لتقسيم 11 على 2 مثلًا، فستُعاد القيمة 5.5. أمَّا في بايثون 2، فحاصل التعبير 11/2 هو 5.

يُجرِي العامل / في بايثون 2 قسمة عملية القسمة مع تقريب الناتج إلى أصغر عدد صحيح له (هذه العملية تدعى القسمة التقريبية [floor division])، إذ أنَّه إن كان حاصل القسمة يساوي x، فسيكون ناتج عملية القسمة في بايثون 2 أكبر عدد من الأعداد الصحيحة الأصغر من أو تساوي x. إذا نفَّذت المثال `print(80 / 5)` أعلاه في بايثون 2 بدلًا من بايثون 3، فسيكون الناتج هو 16، دون الجزء العشري.

في بايثون 3، يمكنك استخدام العامل // لإجراء القسمة التقريبية. التعبير `100 // 40` سيعيد القيمة 2. القسمة التحتية مفيدة في حال كنت تريد أن يكون حاصل القسمة عددًا صحيحًا.

5. عامل باقي القسمة (Modulo)

العامل % هو باقي القسمة (modulo)، والذي يُرجع باقي عملية القسمة. هذا مفيد للعثور على الأعداد التي هي مضاعفات لنفس العدد. المثال التالي يوضح كيفية استخدام عامل الباقي:

```
o = 85
p = 15

print(o % p)    #  10
```

حاصل قسمة 85 على 15 هو 5، والباقي 10. القيمة 10 هي التي ستُعاد هنا لأنَّ عامل الباقي يعيد باقي عملية القسمة.

إذا استخدمنا عددين عشريين مع عامل الباقي، فسيُعاد عدد عشري:

```
q = 36.0
r = 6.0

print(o % p)     #  0.0
```

في حال قسمة 36.0 على 6.0، فلن يكون هناك باقٍ، لذلك تعاد القيمة 0.0.

6. القوة (Power)

يُستخدم عامل القوة ** (يقال له أحيانًا «الأس») في بايثون لرفع العدد الأيسر لقوة الأس للعدد الأيمن. وهذا يعني أنه في التعبير 3 ** 5، العدد 5 سيُرفَع إلى القوة 3. في الرياضيات، غالبًا ما نرى هذا التعبير يُكتب على الشكل 5^3، إذ يُضرب العدد 5 في نفسه 3 مرات. في بايثون، التعبيران 3 ** 5 و 5 * 5 * 5 سيعطيان النتيجة نفسها.

سنستخدم في المثال التالي المتغيرات:

```
s = 52.25
t = 7

print(s ** t)      #  1063173305051.292
```

رفع العدد العشري 52.25 إلى القوة 7 عبر عامل الأسّ ** سينتج عنه عدد عشري كبير.

7. أسبقية العمليات الحسابية

في بايثون، كما هو الحال في الرياضيات، علينا أن نضع في حساباتنا أنَّ المعاملات ستُقيَّم وفقًا لنظام الأسبقية، وليس من اليسار إلى اليمين، أو من اليمين إلى اليسار.

إذا نظرنا إلى التعبير التالي:

```
u = 10 + 10 * 5
```

قد نقرأه من اليسار إلى اليمين، ولكن تذكَّر أنَّ عملية الضرب ستُجرَى أولًا، لذا إن استدعينا `print(u)`، فسنحصل على القيمة التالية:

```
60
```

هذا لأنَّ 10 * 5 ستُقيَّم أولًا، وسينتج عنها العدد 50، ثم يضاف إليها العدد 10 لنحصل على 60 في النهاية.

إذا أردنا بدلًا من ذلك إضافة القيمة 10 إلى 10، ثم ضرب المجموع في 5، فيمكننا استخدام الأقواس كما نفعل في الرياضيات تمامًا:

```
u = (10 + 10) * 5
print(u)     #  100
```

إحدى الطرق البسيطة لتذكر الأسبقيات هي حفظ البيتين التاليين لتذكر أوائل كلماتهما:

> قم أبعد ضيقًا ... قم جد طريقًا

الأسبقية	العملية	الحرف
1	القوس	قم
2	الأس	أبعد
3	الضرب	ضيقًا
4	القسمة	قم
5	الجمع	جد
6	الطرح	طريقًا

8. عامل الإسناد (Assignment Operators)

أكثر عامل الإسناد استخدامًا هو إشارة التساوي =. يُسنِد عامل الإسناد (أو عامل التعيين =) القيمة الموجودة على يمينه إلى المتغير الموضوع على يساره. على سبيل المثال، تُسنِد التعليمة `v = 23` العدد الصحيح 23 للمتغير v.

من الشائع استخدام عاملات الإسناد المركبة التي تجري عملية رياضية على قيمة المتغير، ثم تُسنِد القيمة الجديدة الناتجة إلى ذلك المتغير. تجمع عوامل الإسناد المركبة بين عامل رياضي والعامل =؛ ففي حال الجمع، نستخدم + مع = للحصول على عامل الإسناد المركب +=. لنطبِّق ذلك في مثال عملي:

```
w = 5
w += 1
print(w)    #  6
```

أولاً، نُسنِد القيمة 5 إلى المتغير w، ثم نستخدم معامل الإسناد المركب += لإضافة العدد الصحيح إلى قيمة المتغير الأيسر، ثم نُسنِد النتيجة إلى المتغير w.

تُستخدم معاملات الإسناد المركبة استخدامًا متكررًا مع حلقات for التكرارية، والتي ستستخدمها عندما تريد تكرار عمليةٍ عدة مرات:

```
for x in range (0, 7):
    x *= 2
    print(x)
```

والناتج سيكون:

```
0
2
4
6
8
10
12
```

تمكنا باستخدام الحلقة for من أتمتة العملية *= التي تضرب قيمة المتغير w بالعدد 2، ثم تُسنِد النتيجة إلى المتغير w لأجل استخدامها في التكرار التالي من الحلقة.

لدى بايثون معامل إسناد مركب مقابل لكل من المعاملات الحسابية التي تطرقنا آنفًا إليها وهي:

```
y += 1      # إضافة القيمة ثم إسنادها
y -= 1      # طرح القيمة ثم إسنادها
y *= 2      # ضرب القيمة ثم إسنادها
y /= 3      # تقسيم القيمة ثم إسنادها
y // = 5    # تقسيم سفلي القيمة ثم إسنادها
y **= 2     # تنفيذ عامل الأس على القيمة ثم إسنادها
y %= 3      # إعادة باقي قسمة القيمة ثم إسناده
```

يمكن أن يكون عامل الإسناد المركب مفيدًا عندما تحتاج إلى الزيادة أو الإنقاص التدريجي، أو عندما تحتاج إلى أتمتة عمليات معينة في برنامجك.

9. إجراء العمليات الرياضية عبر الدوال

بعد أن تعرفنا كيفية إجراء العمليات الرياضية عبر العوامل (opreators)، سنستعرض في هذا القسم بعض الدوال الرياضية المُضمَّنة التي يمكن استخدامها مع أنواع البيانات العددية في بايثون 3. فتتضمّن بايثون 3 العديد من الدوال التي يمكنك استخدامها بسهولة في أي برنامج. تتيح لك بعض تلك الدوال تحويل أنواع البيانات، والبعض الآخر خاص بنوع معين، مثل السلاسل النصية.

سنلقي نظرة على الدوال التالية:

- `abs()`: للحصول على القيمة المطلقة
- `divmod()`: للحصول على الحاصل والباقي في وقت واحد
- `pow()`: لرفع عدد لقوة معينة
- `round()`: لتقريب عدد بمنازل عشرية محددة
- `sum()`: لحساب مجموع العناصر في كائن قابلٍ للتكرار (iterable)

فهم هذه الدوال سيمنحك مرونة أكبر في التعامل مع البيانات العددية، ويساعدك على اتخاذ قرارات مدروسة عند تحديد العوامل والدوال التي عليك استخدامها.

ا. القيمة المطلقة

تعيد الدالة المضمّنة `abs()` القيمة المطلقة للعدد الذي يُمرر إليها. في الرياضيات، تشير القيمة المطلقة إلى العدد نفسه إن كانت القيمة موجبة، أو القيمة المعاكسة إن كانت القيمة سالبة. مثلًا، القيمة المطلقة للعدد 15 هي 15، والقيمة المطلقة للعدد ‎-74 هي 74، والقيمة المطلقة للعدد 0 هي 0.

القيمة المطلقة مفهوم مهم في الحساب والتحليل، كما أنَّها مفيدة كذلك في المواقف اليومية، مثل حساب المسافة المقطوعة. على سبيل المثال، إذا كنت تحاول الوصول إلى مكان يبعد 58 ميلًا، ولكنَّك تجاوزت ذلك المكان، وسافرت 93 فرسخًا. فإنَّ حسبت عدد الفراسخ التي ينبغي أن تقطعها الآن للوصول إلى الوجهة المقصودة، فسوف ينتهي بك المطاف بعدد سالب، لكن لا يمكنك السفر عددًا سالبًا من الفراسخ!

سنستخدم الدالة `abs()` لحل هذه المشكلة:

```
# عدد الفراسخ التي تفصلنا عن الوجهة انطلاقًا من المُنطلق
miles_from_origin = 58

# الفراسخ المقطوعة من المُنطلَق إلى الوجهة
miles_travelled = 93

# حساب عدد الفراسخ من الموقع الحالي
miles_to_go = miles_from_origin - miles_travelled

# طباعة عدد الفراسخ المتبقية بعدد سالب
```

```
print(miles_to_go)
# حساب القيمة المطلقة للعدد السالب
print(abs(miles_to_go))
```

المخرجات ستكون:

```
-35
35
```

لولا استخدام الدالة abs()، لحصلنا على عدد سالب، أي ‎-35. ورغم أنَّ miles_travelled أصغر من miles_from_origin، فإنَّ الدالة abs() تحل إشكالية العدد السالب.

عندما نمرِّر لها عددًا سالبًا، ستعيد الدالة abs() عددًا موجبًا، لأنَّ القيمة المطلقة دائمًا تعيد أعدادًا موجبة أو معدومة.

في المثال التالي، سنمرر للدالة abs() عددًا موجبًا، وكذلك الصفر:

```
print(abs(89.9))    #  89.9
print(abs(0))       # 0
```

ب. العثور على الحاصل والباقي بدالة واحدة

القسمة التقريبية (floor division)، التي تُعيد حاصل القسمة [quotient] مع تقريبه)، وقسمة الباقي (modulo division)، التي تعيد باقي القسمة [remainder])، مرتبطان ارتباطًا وثيقًا، وقد يكون من المفيد استخدام دالة تجمع بين العمليتين معًا.

تجمع الدالة المضمَّنة divmod() بين العمليتين، إذ تعيد أولًا حاصل عملية القسمة التقريبية، ثم الباقي. ينبغي تمرير عددين إلى الدالة divmod()، على النحو التالي:

```
divmod(a,b)
```

تكافئ هذه الدالة العمليتين التاليتين:

```
a // b
a & b
```

لنفترض أنّنا كتبنا كتابًا يحتوي 80 ألف كلمة. يريد الناشر أن تحتوي كل صفحة من الكتاب ما بين 300 و 250 كلمة، ونود أن نعرف عدد الصفحات التي ستشكل الكتاب بحسب عدد كلمات الصفحة الذي اخترناه. باستخدام الدالة `divmod()`، يمكننا أن نعرف على الفور عدد الصفحات في الكتاب، وعدد الكلمات المتبقية التي ستُنقل إلى صفحة إضافية.

```
# كم عدد الكلمات في كتابنا
words = 80000

# الخيار A: عدد كلمات الصفحة هو 300
per_page_A = 300

# الخيار B: عدد كلمات الصفحة هو 250
per_page_B = 250

print(divmod(words,per_page_A)) # حساب الخيار A
print(divmod(words,per_page_B)) # حساب الخيار B
```

وسينتج عن هذه الشيفرة:

```
(266, 200)
(320, 0)
```

في الخيار A، سنحصل على 266 صفحة مليئة بالكلمات، و 200 كلمة متبقية (ثلثا صفحة)، والمجموع هو 267 صفحة، وفي الخيار B، سنحصل على كتاب من 320 صفحة. إن أردنا الحفاظ

على البيئة، فالخيار A قد يكون أفضل، ولكن إذا أردنا تصميمًا جذابًا، أو الكتابة بحجم خط كبير، فقد نختار الخيار "B".

تقبل الدالة `divmod()` الأعداد الصحيحة والأعداد العشرية، في المثال التالي سنُمرِّر عددًا عشريًّا إلى الدالة `divmod()`:

```
a = 985.5
b = 115.25

print(divmod(a,b))  #  (8.0, 63.5)
```

في هذا المثال، العدد `8.0` هو حاصل القسمة التحتية للعدد 985.5 مقسومًا على 115.25، و `63.5` هو الباقي.

يمكنك استخدام عامل القسمة التحتية `//` و عامل الباقي `%` للتحقُّق من نتيجة `divmod()`:

```
print(a//b)    #  8.0
print(a%b)    # 63.5
```

ج. القوة (Power)

في بايثون، يمكنك استخدام عامل القوة `**` (أو الأس) لرفع عدد إلى قوة معينة، أو يمكنك استخدام الدالة `pow()` المضمَّنة التي تأخذ عددين وتجري العملية نفسها.

لتوضيح كيفيَّة عمل الدالة `pow()`، لنقل أنَّنا نجري أبحاثًا على البكتيريا، ونريد أن نقدِّر عدد البكتيريا التي سنحصل عليها في نهاية اليوم إذا بدأنا ببكتيريا واحدة.

لنفترض أنَّ البكتيريا التي نعمل عليها تتضاعف في كل ساعة، لذلك النتيجة النهائية ستكون 2 قوة العدد الكلي لعدد الساعات التي مرت (24 في حالتنا).

```
hours = 24
total_bacteria = pow(2,hours)

print(total_bacteria)    # 16777216
```

لقد مرّرنا عددين صحيحين للدالة `pow()`، والنتيجة التي حصلنا عليها، والتي تمثِّل عدد البكتيريا بحلول نهاية اليوم، هي أكثر من 16 مليون بكتيريا.

في الرياضيات، نكتب "3 أسّ 3" بشكل عام على بالشكل 3^3 والتي تكافئ $3 \times 3 \times 3$، أي 27. ولحساب 3^3 في بايثون، نكتب `pow(3,3)`، إذ تقبل الدالة `pow()` الأعداد الصحيحة والأعداد العشرية، وتوفِّر بديلًا لعامل الأس `**`.

د. تقريب الأعداد

تقريب الأعداد (Rounding Numbers) ضروري عند العمل مع الأعداد العشرية التي تحتوي على الكثير من المنازل (الأجزاء) العشرية. تقبل الدالة المضمنة `round()` عددين: أحدها يمثِّل العدد المراد تقريبه والآخر يحدِّد عدد المنازل العشرية المراد الإبقاء عليها (أي قيمة التقريب).

سنستخدم هذه الدالة لتقريب عدد عشري له أكثر من 10 منازل عشرية والحصول على عدد بأربعة منازل عشرية فقط:

```
i = 17.34989436516001
print(round(i,4))    # 17.3499
```

في المثال أعلاه، تم تقريب العدد `17.34989436516001` إلى `17.3499` لأنَّنا حدَّدنا عدد المنازل العشرية التي ينبغي الاقتصار عليها بأربعة.

لاحظ أيضًا أنَّ الدالة `round()` تقرِّب الأعداد إلى الأعلى، لذا بدلًا من إعادة 17.3498، فقد أعادت 17.3499، لأنَّ الرقم الذي يلي المنزل العشري 8 هو الرقم 9. وسيقرَّب أي عدد متبوع بالعدد 5 أو أكبر إلى العدد الصحيح التالي.

في الحياة اليومية، نُقرِّب الأعداد لأسباب كثيرة، وخاصة في التعاملات الماليَّة؛ فلا يمكننا تقسيم فلس واحد بالتساوي بين عدة أصدقاء مثلا.

سنكتب في المثال التالي برنامجا بسيطا يمكنه حساب البقشيش، إذ سنحدِّد فيه قيم المتغيرات، ولكن يمكنك إعادة كتابة البرنامج لجعل المستخدمين يدخلون القيم بأنفسهم. في هذا المثال، ذهب 3 أصدقاء إلى مطعم، وأرادوا تقسيم الفاتورة، والتي تبلغ 87.93 دولارًا بالتساوي، بالإضافة إلى إكرامية (بقشيش) بنسبة 20٪:

```
bill = 87.93             # إجمالي الفاتورة
tip = 0.2                # ٪ 20 بقشيش
split = 3                # عدد الناس الذين سيتشاركون الفاتورة

# حساب الفاتورة الإجمالية
total = bill + (bill * tip)
# حساب ما ينبغي أن يدفعه كل شخص
each_pay = total / split
# ما ينبغي أن يدفعه كل شخص قبل التقريب
print(each_pay)
# تقريب العدد فلا يمكننا تقسيم الفلسات
print(round(each_pay,2))
```

والمخرجات ستكون:

```
35.172000000000004
35.17
```

في هذا البرنامج، نطلب أولًا إخراج العدد بعد حساب إجمالي الفاتورة والإكراميات مقسومًا على 3، النتيجة ستكون عددًا يتضمَّن الكثير من المنازل العشرية: 35.172000000000004. نظرًا لأنَّ هذا العدد لا يمثِّل مبلغًا ماليًا واقعيًّا، فإنَّنا نستخدم الدالة round()، ونقرِّب المنازل العشرية على 2، حتى نتمكَّن من توفير ناتج يمكن للأصدقاء الثلاثة أن يدفعوه: 35.17.

إذا كنت تفضل التقريب إلى عدد بلا منازل عشرية، يمكنك تمرير 0 كمعامل ثان إلى الدالة round():

```
round(345.9874590348545304636,0)
```

القيمة الناتجة ستكون 346.0.

يمكنك أيضًا تمرير الأعداد الصحيحة إلى round() دون الخوف من تلقي خطأ، وهذا مفيد في حال تلقيت من المستخدم عددًا صحيحًا بدلًا من عدد عشري. وفي هذه الحالة، سيُعاد عدد صحيح.

ه. حساب المجموع

تُستخدَم الدالة sum() لحساب مجاميع أنواع البيانات العدديَّة المركَّبة (numeric compound data types)، بما في ذلك القوائم، والصفوف، والقواميس.

يمكننا تمرير قائمة إلى الدالة sum() لجمع كل عناصرها بالترتيب من اليسار إلى اليمين:

```
some_floats = [1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9]
print(sum(some_floats))     #  49.5
```

نفس النتيجة سنحصل عليها إن استخدمنا الصفوف والقواميس:

```
# حساب مجموع الأعداد في الصف
print(sum((8,16,64,512)))
# حساب مجموع الأعداد في القاموس
print(sum({-10: 'x', -20: 'y', -30: 'z'}))
```

المخرجات:

```
60
-60
```

يمكن أن تأخذ الدالة `sum()` وسيطين، الوسيط الثاني سيُضاف إلى المجموع الناتج:

```
some_floats = [1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9]

print(sum(some_floats, 0.5))          # 50.0
print(sum({-10: 'x', -20: 'y', -30: 'z'},60))   # 0
```

القيمة الافتراضية للوسيط الثاني هي 0.

10. خلاصة الفصل

غطّينا في هذا الفصل العديد من العوامل والدوال التي ستستخدمها مع الأعداد الصحيحة والعشرية لإجراء أهم العمليات الرياضية ولا زال هنالك الكثير منها، فننصحك بزيارة توثيق العمليات العددية في بايثون وتوثيق الأنواع العددية في بايثون في موسوعة حسوب لمزيد من التفاصيل.

10

العمليات المنطقية (البوليانية)

هناك قيمتان فقط لنوع البيانات المنطقية، وهما `True` و `False`. تُستخدَم الحسابات المنطقية في البرمجة لإجراء الموازنات، والتحكم في مسار البرنامج.

تمثِّل القيم المنطقية قيم الحقيقة (truth values) في علم المنطق في الرياضيات، وتُكتَب القيمتان `True` و `False` دائمًا بالحرفين الكبيرين T و F على التوالي، لأنَّهما قيمتان خاصتان في بايثون. سنتعرف في هذا الفصل على العمليات المنطقيَّة في بايثون، بما في ذلك الموازنة المنطقيَّة، والعوامل المنطقيَّة، وجداول الحقيقة.

1. عامل الموازنة

في البرمجة، تُستخدَم عاملات الموازنة (comparison operators) للموازنة بين القيم، وتعيد إحدى القيمتين المنطقتين `True` و `False`. يوضح الجدول أدناه عاملات الموازنة المنطقية:

العامل	الشرح
`==`	يساوي
`!=`	يخالف
`<`	أصغر من
`>`	أكبر من
`<=`	أصغر من أو يساوي
`>=`	أكبر من أو يساوي

لفهم كيفية عمل هذه العاملات، سنستخدم المتغيرين التاليين:

```
x = 5
y = 8
```

في هذا المثال، لمَّا كان x يساوي 5، فهو أصغر من y ذي القيمة 8.

باستخدام هذين المتغيرين والقيم المرتبطة بهما، سنجرِّب أحد العاملات من الجدول أعلاه. سنطلب من بايثون أن تطبع ناتج عملية الموازنة، إمَّا True أو False. لتوضيح المثال أكثر، سنطبع سلسلة نصية لتوضيح ما جرى تقييمه.

```
x = 5
y = 8

print("x == y:", x == y)
print("x != y:", x != y)
print("x < y:", x < y)
print("x > y:", x > y)
print("x <= y:", x <= y)
print("x >= y:", x >= y)
```

والمخرجات هي:

```
x == y: False
x != y: True
x < y: True
x > y: False
x <= y: True
x >= y: False
```

باتباع المنطق الرياضي، في كل من التعبيرات المذكورة أعلاه، هذه نتيجة الموازنات:

- 5 (قيمة x) تساوي 8 (قيمة y)؟ False، خطأ

- 5 تخالف 8؟ True، صحيح
- 5 أصغر من 8؟ True، صحيح
- 5 أكبر من 8؟ False، خطأ
- 5 أصغر من أو يساوي 8؟ True، صحيح
- 5 ليس أصغر من أو يساوي 8؟ False، خطأ

رغم استخدامنا للأعداد الصحيحة هنا، إلا أنَّه بإمكاننا استبدال الأعداد العشرية بها.

يمكن أيضًا استخدام السلاسل النصية مع المعاملات المنطقية. وهي حساسة لحالة الأحرف، ما لم تستخدم تابعًا إضافيًا للسلاسل النصية، ويوضح المثال التالي كيفية موازنة السلاسل النصية:

```
Sammy = "Sammy"
sammy = "sammy"

print("Sammy == sammy: ", Sammy == sammy)      # Sammy == sammy: 
False
```

السلسلة "Sammy" أعلاه لا تساوي السلسلة النصية "sammy"، لأنَّهما ليستا متماثلتين تمامًا؛ فإحداهما تبدأ بحرف كبير S، والأخرى بحرف صغير s. ولكن لو أضفنا متغيرًا آخر قيمته "Sammy"، فستكونان متساويتين:

```
Sammy = "Sammy"
sammy = "sammy"
also_Sammy = "Sammy"
```

```
print("Sammy == sammy: ", Sammy == sammy)
# Sammy == sammy:  False
print("Sammy == also_Sammy", Sammy == also_Sammy)
# Sammy == also_Sammy:  True
```

يمكنك أيضًا استخدام عاملات الموازنة الأخرى، بما في ذلك ‎>‎ و ‎<‎ لموازنة سلسلتين نصيتين. ستوازن بايثون هذه السلاسل النصية بحسب الترتيب المعجمي في نظام محارف ASCII. ويمكنك أيضًا تقييم القيم المنطقية باستخدام عاملات الموازنة:

```
t = True
f = False

print("t != f: ", t != f)    #  t != f:  True
```

تبيِّن الشيفرة البرمجية أعلاه أنَّ True لا تساوي False.

لاحظ الفرق بين العاملين = و ==.

```
x = y   # إسناد قيمة y إلى x
x == y  # تتحقق مما إذا كان y و x متساويين
```

الأول =، هو عامل الإسناد (assignment operator)، والذي سيحدِّد قيمة أحد المتغيرين، ويجعلها مساوية لقيمة الآخر. الثاني ==، وهو عامل الموازنة الذي سيحدِّد ما إذا كانت القيمتان متساويتين.

2. العاملات المنطقية

هناك ثلاثة عاملات منطقية (Boolean operators) تُستخدم لموازنة القيم. وتعيد إمَّا True أو False. هذه العاملات هي، and (و)، و or (أو)، و not (النفي)، وقد عرَّفناها في الجدول أدناه.

العامل	الشرح	الصياغة
and	إن كان كلا التعبيرين صحيحين True	x and y
or	إن كان أحد التعبيرين على الأقل صحيحًا True	x or y
not	إن كان التعبير خطأ True	not x

عادةً ما تُستخدَم العاملات المنطقية لتقييم ما إذا كان تعبيران منطقيان صحيحين أم لا. على سبيل المثال، يمكن استخدامها لتحديد ما إذا كان الطالب قد نجح، وأنَّه مُسجل في الفصل، وإذا كانت كلتا الحالتان صحيحتين، فسيُضَاف الطالب إلى سجل النظام. مثال آخر هو تحديد ما إذا كان المستخدم عميلًا نشطًا لمتجر إلكتروني استنادًا إلى ما إذا كان لديه رصيد في المتجر، أو أنَّه اشترى خلال الأشهر الستة الماضية.

لفهم كيفيَّة عمل العاملات المنطقية، دعنا نقيِّم التعابير الثلاث التالية:

```
print((9 > 7) and (2 < 4))   # كلا التعبيرين صحيحان
print((8 == 8) or (6 != 6))  # أحد التعبيرين صحيح
print(not(3 <= 1))           # التعبير الأصلي خطأ
```

والمخرجات هي:

```
True
True
True
```

في الحالة الأولى، `print((9 > 7) and (2 < 4))`، ينبغي أن يكون كلا التعبيرين `7 > 9` و `4 < 2` صحيحين لتكون النتيجة `True` لأنَّ المعامل `and` مُستخدَم.

في الحالة الثانية، `print((8 == 8) or (6 != 6))`، بما أنَّ قيمة `8 == 8` قُيِّمت إلى `True`، فلا يهم ما يقيَّم إليه التعبير `6 != 6`. لكن لو استخدمنا العامل `and`، لقُيِّم التعبير إلى القيمة `False`.

في الحالة الثالثة، `print(not(3 <= 1))`، ينفي العامل `not` القيمة `False` التي تعيدها ناتج العملية المنطقية `3 <= 1`.

في المثال التالي، سنستبدل بالأعداد العشرية أعدادًا صحيحة:

```
print((-0.2 > 1.4) and (0.8 < 3.1))     # أحد التعبيرين خطأ
print((7.5 == 8.9) or (9.2 != 9.2))     # كلا التعبيرين خطأ
print(not(-5.7 <= 0.3))                 # التعبير الأصلي صحيح
```

في المثال أعلاه:

- `and`: يجب أن يكون واحد على الأقل من التعبيرين خطأ ليعيد القيمة `False`،
- `or`: يجب أن يكون كلا التعبيرين خطأ لتعيد القيمة `False`،
- `not`: يجب أن يكون التعبير المرافق له صحيحًا حتى يعيد القيمة `False`.

إذا لم تكن النتائج أعلاه واضحة، فسنعرض بعض جداول الحقيقة أدناه لتفهم الأمر فهمًا أفضل.

يمكنك أيضًا كتابة عبارات مُركَّبة باستخدام `and`، و `or`، و `not`:

```
not((-0.2 > 1.4) and ((0.8 < 3.1) or (0.1 == 0.1)))
```

التعبير الداخلي: `(0.1 == 0.1) or (3.1 > 0.8)` يعيد القيمة True، لأنَّ كلا التعبيرين الرياضيين محقَّقان، أي يعيدان True.

الآن، نأخذ القيمة المُعادة True ونجمعها مع التعبير المنطقي التالي:

```
(-0.2> 1.4) and (True)
```

هذا المثال يعيد False، لأنَّ التعبير `-0.2 > 1.4` يقيَّم إلى القيمة False، و `(False) and (True)` ينتج عنه القيمة False.

أخيرًا، يعيد التعبير الخارجي: `not (False)` القيمة True، وبالتالي ستكون القيمة النهائية المعادة هي:

```
True
```

3. جداول الحقيقة (Truth Tables)

المنطق مجال واسع، وسنكتفي في هذا الفصل ببعض الأفكار المهمَّة التي يمكن أن تساعدك على تحسين طريقة تفكيرك وخوارزمياتك.

فيما يلي جداول الحقيقة لمعامل الموازنة ==، والمعاملات المنطقية and, و or, و not. من المفيد أن تحفظ كيفيَّة عملها، فذلك سيجعلك أسرع في اتخاذ القرارات أثناء كتابة الشيفرات البرمجية.

ا. جدول الحقيقة الخاص بالعامل ==

x	==	y	القيمة المُعادة
True	==	True	True
True	==	False	False
False	==	True	False
False	==	False	True

ب. جدول الحقيقة الخاص بالعامل AND

x	and	y	القيمة المُعادة
True	and	True	True
True	and	False	False
False	and	True	False
False	and	False	False

ج. جدول الحقيقة الخاص بالعامل OR

x	or	y	القيمة المُعادة
True	or	True	True
True	or	False	True
False	or	True	True
False	or	False	False

د. جدول الحقيقة الخاص بالعامل NOT

x	not	القيمة المُعادة
True	not	False
False	not	True

تُستخدَم جداول الحقيقة في المنطق الرياضياتي كثيرًا، وهي مفيدة، ويجب حفظها ووضعها في الحسبان عند إنشاء الخوارزميات البرمجية.

4. استعمال المنطق للتحكم في مسار البرنامج

للتحكم في مسار ونتائج البرنامج عبر التعليمات الشرطيَّة (flow control statements)، يمكننا استخدام «شرط» (condition) متبوعًا «بعبارة برمجية» (clause).

يُقيَّم الشرط بإحدى القيمتين True أو False، و تُستخدَم تلك القيمة في اتخاذ قرار في البرنامج. أمَّا العبارة (clause) فهي الكتلة البرمجية التي تعقب الشرط وتحدِّد نتيجة البرنامج وما ينبغي فعله في حال تحقق الشرط أو عدمه.

تُظهر الشيفرة أدناه مثالًا على معاملات الموازنة التي تعمل مع العبارات الشرطية للتحكم في مسار البرنامج:

```
if grade >= 65:                # شرط
    print("Passing grade")     # بند

else:
    print("Failing grade")
```

سيحدِّد هذا البرنامج ما إذا كان الطالب سينجح أم يرسب. في حال كانت الدرجة التي حصل عليها الطالب تساوي 83 مثلًا، تقييم العبارة الأولى سيكون `True`، وسيُطبَع النص `"Passing grade"`. أمّا إن كانت درجة الطالب هي 59، فتقييم العبارة الأولى سيكون `False`، وبالتالي سينتقل البرنامج لتنفيذ التعليمة المرتبطة بالفرع `else`، أي سيطبع `"Failing grade"`.

يمكن تقييم كل كائنات بايثون بإحدى القيمتين `True` أو `False`، لذلك يوصي الدليل PEP 8 بعدم موازنة كائن بإحدى القيمتين `True` أو `False`، لأنّ ذلك قد يؤدي إلى إعادة قيم منطقية غير متوقَّعة. على سبيل المثال، عليك تجنب استخدام مثل هذا التعبير `sammy == True` في برامجك.

تساعد العاملات المنطقيّة على صياغة شروط يمكن استخدامها لتحديد النتيجة النهائية للبرنامج من خلال التحكم في مسار التنفيذ.

5. خلاصة الفصل

ألقينا في هذا الفصل نظرة على الموازنات والعاملات المنطقية، بالإضافة إلى جداول الحقيقة، وكيفية استخدام القيم المنطقية للتحكم في مسار البرنامج.

11

النوع List: مدخل إلى القوائم

النـوع `list` (القائمة) هي بنيـة بيانـات في بـايثون، وهي عبـارة عن تسلسـل مُـرتَّب قابـل للتغيير من تجميعة عناصر. سنتعرف في هذا الفصل على القوائم وتوابعها وكيفيَّة استخدامها.

القوائم مناسبة لتجميع العناصر المترابطة، إذ تمكِّنك من تجميع البيانات المتشابهة، أو التي تخدم غرضًا معيَّنا معًا، وترتيب الشيفرة البرمجية، وتنفيذ التوابع والعمليات على عدة قيم في وقت واحد.

تساعد القوائم في بايثون وغيرها من هياكل البيانات المركبة على تمثيل التجميعات، مثل تجميعـة ملفـات في مجلـد على حاسـوبك، أو قـوائم التشـغيل، أو رسـائل البريـد الإلكـتروني وغير ذلك.

في المثال التالي، سننشئ قائمةً تحتوي على عناصر من نوع السلاسل النصية:

```
sea_creatures = ['shark', 'cuttlefish', 'squid', 'mantis
shrimp', 'anemone']
```

عندما نطبع القائمة، فستشبه المخرجات القائمة التي أنشأناها:

```
print(sea_creatures)
```

الناتج:

```
['shark', 'cuttlefish', 'squid', 'mantis shrimp', 'anemone']
```

بوصـفها تسلسـلًا مرتبًـا من العناصـر، يمكن اسـتدعاء أيِّ عنصـر من القائمـة عـبر الفهرسـة. القوائم هي بيانات مركَّبة تتألف من أجزاء أصغر، وتتميز بالمرونة، إذ يمكن إضافة عناصر إليها، وإزالتها، وتغييرها. عندما تحتاج إلى تخزين الكثير من القيم، أو التكرار (iterate) عليها، وتريد أن تملك القدرة على تعديل تلك القيم بسهولة، فالقوائم هي خيارك الأفضل.

1. فهرسة القوائم (Indexing Lists)

كل عنصر في القائمة يقابله رقم يمثل فهرسَ ذلك العنصر، والذي هو عدد صحيح، فهرس العنصر الأول في القائمة هو 0.

إليك تمثيل لفهارس القائمة sea_creatures:

anemone	shrimp	squid	shark	shark
4	3	2	1	0

يبدأ العنصر الأول، أي السلسلة النصية shark، عند الفهرس 0، وتنتهي القائمة عند الفهرس 4 الذي يقابل العنصر anemone.

نظرًا لأنَّ كل عنصر في قوائم بايثون يقابله رقم فهرس، يمكننا الوصول إلى عناصر القوائم ومعالجتها كما نفعل مع أنواع البيانات المتسلسلة الأخرى.

يمكننا الآن استدعاء عنصر من القائمة من خلال رقم فهرسه:

```
print(sea_creatures[1])     #  cuttlefish
```

تتراوح أرقام الفهارس في هذه القائمة بين 0 و 4، كما هو موضَّح في الجدول أعلاه. المثال التالي يوضِّح ذلك:

```
sea_creatures[0] = 'shark'
sea_creatures[1] = 'cuttlefish'
sea_creatures[2] = 'squid'
sea_creatures[3] = 'mantis shrimp'
sea_creatures[4] = 'anemone'
```

إذا استدعينا القائمة sea_creatures برقم فهرس أكبر من 4، فسيكون الفهرس خارج النطاق، وسيُطلَق الخطأ IndexError:

```
print(sea_creatures[18])
```

والمخرجات ستكون:

```
IndexError: list index out of range
```

يمكننا أيضًا الوصول إلى عناصر القائمة بفهارس سالبة، والتي تُحسَب من نهاية القائمة، بدءًا من 1-. هذا مفيد في حال كانت لدينا قائمة طويلة، وأردنا تحديد عنصر في نهايته.

بالنسبة للقائمة sea_creatures، تبدو الفهارس السالبة كما يلي:

anemone	shrimp	squid	cuttlefish	shark
-1	-2	-3	-4	-5

في المثال التالي، سنطبع العنصر squid باستخدام فهرس سالب:

```
print(sea_creatures[-3])    #  squid
```

يمكننا ضم (concatenate) سلسلة نصية مع سلسلة نصية أخرى باستخدام العامل +:

```
print('Sammy is a ' + sea_creatures[0])    # Sammy is a shark
```

لقد ضممنا السلسلة النصية Sammy is a مع العنصر ذي الفهرس 0. يمكننا أيضًا استخدام المعامل + لضمِّ قائمتين أو أكثر معًا (انظر الفقرة أدناه).

تساعدنا الفهارس على الوصول إلى أيِّ عنصر من عناصر القائمة والعمل عليه.

2. تعديل عناصر القائمة

يمكننا استخدام الفهرسة لتغيير عناصر القائمة، عن طريق إسناد قيمة إلى عُنصر مُفهرس من القائمة. هذا يجعل القوائم أكثر مرونة، ويسهّل تعديل وتحديث عناصرها.

إذا أردنا تغيير قيمة السلسلة النصية للعنصر الموجود عند الفهرس 1، من القيمة cuttlefish إلى octopus، فيمكننا القيام بذلك على النحو التالي:

```
sea_creatures[1] = 'octopus'
```

الآن، عندما نطبع sea_creatures، ستكون النتيجة:

```
print(sea_creatures)    # ['shark', 'octopus', 'squid', 'mantis
shrimp', 'anemone']
```

يمكننا أيضًا تغيير قيمة عنصر باستخدام فهرس سالب:

```
sea_creatures[-3] = 'blobfish'
print(sea_creatures)      #   ['shark', 'octopus', 'blobfish',
'mantis shrimp', 'anemone']
```

الآن استبدلنا بالسلسلة squid السلسلة النصية blobfish الموجودة عند الفهرس السالب 3- (والذي يقابل الفهرس الموجب 2).

3. تقطيع القوائم (Slicing Lists)

يمكننا أيضا استدعاء عدة عناصر من القائمة. لنفترض أنّنا نرغب في طباعة العناصر الموجودة في وسط القائمة sea_creatures، يمكننا القيام بذلك عن طريق اقتطاع شريحة (جزء) من القائمة.

يمكننا باستخدام الشرائح استدعاء عدة قيم عن طريق إنشاء مجال من الفهارس مفصولة بنقطتين [x: y]:

```
print(sea_creatures[1:4])    # ['octopus', 'blobfish', 'mantis shrimp']
```

عند إنشاء شريحة، كما في [1: 4]، يبدأ الاقتطاع من العنصر ذي الفهرس الأول (مشمولا)، وينتهي عند العنصر ذي الفهرس الثاني (غير مشمول)، لهذا طُبعَت في مثالنا أعلاه العناصر الموجودة في المواضع، 1، و 2، و 3.

إذا أردنا تضمين أحد طرفي القائمة، فيمكننا حذف أحد الفهرسَين في التعبير [x: y]. على سبيل المثال، إذا أردنا طباعة العناصر الثلاثة الأولى من القائمة sea_creatures، يمكننا فعل ذلك عن طريق كتابة:

```
print(sea_creatures[:3])    # ['shark', 'octopus', 'blobfish']
```

لقد طُبعَت العناصر من بداية القائمة حتّى العنصر ذي الفهرس 3. لتضمين جميع العناصر الموجودة في نهاية القائمة، سنعكس الصياغة:

```
print(sea_creatures[2:])    # ['blobfish', 'mantis shrimp', 'anemone']
```

يمكننا أيضًا استخدام الفهارس السالبة عند اقتطاع القوائم، تمامًا كما هو الحال مع الفهارس الموجبة:

```
print(sea_creatures[-4:-2])        # ['octopus', 'blobfish']
print(sea_creatures[-3:])          # ['blobfish', 'mantis shrimp', 'anemone']
```

هناك معامل آخر يمكننا استخدامه في الاقتطاع، ويشير إلى عدد العناصر التي يجب أن تخطيها (الخطوة) بعد استرداد العنصر الأول من القائمة. حتى الآن، لقد أغفلنا المعامل stride، وستعطيه بايثون القيمة الافتراضية 1، ما يعني أنَّه سيتم استرداد كل العناصر الموجودة بين الفهرسَين المحددين.

ستكون الصياغة على الشكل التالي [x: y: z]، إذ يشير z إلى الخطوة (stride). في المثال التالي، سننشئ قائمة كبيرة، ثم نقتطعها، مع خطوة اقتطاع تساوي 2:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

print(numbers[1:11:2])        #  [1, 3, 5, 7, 9]
```

سيطبع التعبير [2 :11 :numbers[1 القيم ذات الفهارس المحصورة بين 1 (مشمولة) و 11 (غير مشمولة)، وسيقفز البرنامج بخطوتين كل مرة، ويطبع العناصر المقابلة.

يمكننا حذف المُعاملين الأوَّليين، واستخدام الخطوة وحدها بعدِّها معاملًا وفق الصياغة التالية: [::z]:

```
print(numbers[::3])      #  [0, 3, 6, 9, 12]
```

عند طباعة القائمة numbers مع تعيين الخطوة عند القيمة 3، فلن تُطبَع إلا العناصر التي فهارسها من مضاعفات 3. يجعل استخدام الفهارس الموجبة والسالبة ومعامل الخطوة في اقتطاع القوائم التحكم في القوائم ومعالجتها أسهل وأكثر مرونة.

4. تعديل القوائم بالعوامل

يمكن استخدام العوامل لإجراء تعديلات على القوائم. سننظر في استخدام العاملين + و * ومقابليهما المركبين += و=*.

يمكن استخدام العامل + لضمِّ (concatenate) قائمتين أو أكثر معًا:

```
sea_creatures = ['shark', 'octopus', 'blobfish', 'mantis
shrimp', 'anemone']
oceans = ['Pacific', 'Atlantic', 'Indian', 'Southern',
'Arctic']

print(sea_creatures + oceans)
```

والمخرجات هي:

```
['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone',
'Pacific', 'Atlantic', 'Indian', 'Southern', 'Arctic']
```

يمكن استخدام العامل + لإضافة عنصر (أو عدة عناصر) إلى نهاية القائمة، لكن تذكر أن تضع العنصر بين قوسين مربعين:

```
sea_creatures = sea_creatures + ['yeti crab']
print (sea_creatures)    #    ['shark', 'octopus', 'blobfish',
'mantis shrimp', 'anemone', 'yeti crab']
```

يمكن استخدام العامل * لمضاعفة القوائم (multiply lists). ربما تحتاج إلى عمل نُسِخٍ لجميع الملفات الموجودة في مجلد على خادم، أو مشاركة قائمة أفلام مع الأصدقاء؛ ستحتاج في هذه الحالات إلى مضاعفة مجموعات البيانات.

سنضاعف القائمة sea_creatures مرتين، والقائمة oceans ثلاث مرات:

```
print(sea_creatures * 2)
print(oceans * 3)
```

والنتيجة ستكون:

```
['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone',
'yeti crab', 'shark', 'octopus', 'blobfish', 'mantis shrimp',
'anemone', 'yeti crab']
['Pacific', 'Atlantic', 'Indian', 'Southern', 'Arctic',
'Pacific', 'Atlantic', 'Indian', 'Southern', 'Arctic',
'Pacific', 'Atlantic', 'Indian', 'Southern', 'Arctic']
```

يمكننا باستخدام العامل * نسخ القوائم عدة مرات.

يمكننا أيضًا استخدام الشكلين المركبين للعاملين + و * مع عامل الإسناد =. يمكن استخدام العاملين المركبين += و *= لملء القوائم بطريقة سريعة ومُؤتمتة. يمكنك استخدام هذين العاملين لملء القوائم بعناصر نائبة (placeholders) يمكنك تعديلها في وقت لاحق بالمدخلات المقدمة من المستخدم على سبيل المثال.

في المثال التالي، سنضيف عنصرًا إلى القائمة sea_creatures. سيعمل هذا العنصر مثل عمل العنصر النائب، ونود إضافة هذا العنصر النائب عدة مرات. لفعل ذلك، سنستخدم العامل += مع الحلقة for.

```
for x in range(1,4):
    sea_creatures += ['fish']
    print(sea_creatures)
```

والمخرجات ستكون:

```
['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone', 
'yeti crab', 'fish']
['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone', 
'yeti crab', 'fish', 'fish']
['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone', 
'yeti crab', 'fish', 'fish', 'fish']
```

سيُضاف لكل تكرار في الحلقة for عنصر fish إلى القائمة sea_creatures.

يتصرف العامل =* بطريقة مماثلة:

```
sharks = ['shark']

for x in range(1,4):
    sharks *= 2
    print(sharks)
```

الناتج سيكون:

```
['shark', 'shark']
['shark', 'shark', 'shark', 'shark']
['shark', 'shark', 'shark', 'shark', 'shark', 'shark', 'shark', 
'shark']
```

5. إزالة عنصر من قائمة

يمكن إزالة العناصر من القوائم باستخدام del. سيؤدي ذلك إلى حذف العنصر الموجود عند الفهرس المحدد.

سنزيل من القائمة sea_creatures العنصر octopus. هذا العنصر موجود عند الفهرس 1.

لإزالة هذا العنصر، سنستخدم del ثم نستدعي متغير القائمة وفهرس ذلك العنصر:

```
sea_creatures =['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone', 'yeti crab']

del sea_creatures[1]
print(sea_creatures)     #  ['shark', 'blobfish', 'mantis shrimp', 'anemone', 'yeti crab']
```

الآن، العنصر ذو الفهرس 1، أي السلسلة النصية octopus، لم يعد موجودًا في قائمتنا.

يمكننا أيضًا تحديد مجال مع العبارة del. لنقل أنَّنا نريد إزالة العناصر octopus، و blobfish و mantis shrimp معًا. يمكننا فعل ذلك على النحو التالي:

```
sea_creatures =['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone', 'yeti crab']

del sea_creatures[1:4]
print(sea_creatures)     #  ['shark', 'anemone', 'yeti crab']
```

باستخدام مجال مع del، تمكّنا من إزالة العناصر الموجودة بين الفهرسَين 1 (مشمول) و 4 (غير مشمول)، والقائمة أضحت مكوَّنة من 3 عناصر فقط بعد إزالة 3 عناصر منها.

6. بناء قوائم من قوائم أخرى موجودة

يمكن أن تتضمَّن عناصر القوائم قوائم أخرى، مع إدراج كل قائمة بين قوسين معقوفين داخل الأقواس المعقوفة الخارجية التابعة لقائمة الأصلية:

```
sea_names = [['shark', 'octopus', 'squid', 'mantis shrimp'], ['Sammy', 'Jesse', 'Drew', 'Jamie']]
```

تسمى القوائم المُتضمّنة داخل قوائم أخرى بالقوائم المتشعبة (nested lists).

للوصول إلى عنصر ضمن هذه القائمة، سيتعيَّن علينا استخدام فهارس متعددة تقابل

مستوى التشعُّب:

```
print(sea_names[1][0])    #  Sammy
print(sea_names[0][0])    #  shark
```

فهرس القائمة الأولى يساوي 0، والقائمة الثانية فهرسُها 1. ضمن كل قائمة متشعِّبة داخلية، سيكون هناك فهارس منفصلة، والتي سنسميها فهارس ثانوية:

```
sea_names[0][0] = 'shark'
sea_names[0][1] = 'octopus'
sea_names[0][2] = 'squid'
sea_names[0][3] = 'mantis shrimp'

sea_names[1][0] = 'Sammy'
sea_names[1][1] = 'Jesse'
sea_names[1][2] = 'Drew'
sea_names[1][3] = 'Jamie'
```

عند العمل مع قوائم مؤلَّفة من قوائم، من المهم أن تعي أنَّك ستحتاج إلى استخدام أكثر من فهرس واحد للوصول إلى عناصر القوائم المتشعبة.

7. استخدام توابع القوائم

سنَتعرَّف الآن على التوابع المُضمَّنة التي يمكن استخدامها مع القوائم. ونتعلَّم كيفيَّة إضافة عناصر إلى القائمة وكيفيَّة إزالتها، وتوسيع القوائم وترتيبها، وغير ذلك.

القوائم أنواعٌ قابلةٌ للتغيير (mutable) على عكس السلاسل النصية التي لا يمكن تغييرها، فعندما تستخدم تابعًا على قائمة ما، ستؤثر في القائمة نفسها، وليس في نسخة منها.

سنعمل في هذا القسم على قائمة تمثِّل حوض سمك، إذ ستحوي القائمة أسماء أنواع الأسماك الموجودة في الحوض، وسنعدلها كلمَّا أضفنا أسماكًا أو أزلناها من الحوض.

ا. التابع `list.append()`

يضيف التابع `list.append(x)` عنصرًا (العنصر x الممرّر) إلى نهاية القائمة list.

يُعرِّف المثال التالي قائمةً تمثل الأسماك الموجودة في حوض السمك.

```
fish = ['barracuda','cod','devil ray','eel']
```

تتألف هذه القائمة من 4 سلاسل نصية، وتتراوح فهارسها من 0 إلى 3.

سنضيف سمكة جديدة إلى الحوض، ونود بالمقابل أن نضيف تلك السمكة إلى قائمتنا. سنمرِّر السلسلة النصية flounder التي تمثِّل نوع السمكة الجديدة إلى التابع `list.append()`، ثم نطبع قائمتنا المعدلة لتأكيد إضافة العنصر:

```
fish.append('flounder')
print(fish)
#  ['barracuda', 'cod', 'devil ray', 'eel', 'flounder']
```

الآن، صارت لدينا قائمة من 5 عناصر، تنتهي بالعنصر الذي أضفناه للتو عبر التابع `append()`.

ب. التابع `list.insert()`

يأخذ التابع `list.insert (i,x)` وسيطين: الأول i يمثِّل الفهرس الذي ترغب في إضافة العنصر عنده، و x يمثل العنصر نفسه.

لقد أضفنا إلى حوض السمك سمكة جديدة من نوع anchovy. ربما لاحظت أنَّ قائمة الأسماك مرتبة ترتيبًا أبجديًا حتى الآن، لهذا السبب لا نريد إفساد الترتيب، ولن نضيف السلسلة النصية anchovy إلى نهاية القائمة باستخدام الدالة `list.append()`؛ بدلًا من ذلك، سنستخدم التابع `list.insert()` لإضافة anchovy إلى بداية القائمة، أي عند الفهرس 0:

```
fish.insert(0,'anchovy')
print(fish)
#  ['anchovy', 'barracuda', 'cod', 'devil ray', 'eel',
'flounder']
```

في هذه الحالة، أضفنا العنصر إلى بداية القائمة.

ستتقدم فهارس العناصر التالية خطوةً واحدةً إلى الأمام. لذلك، سيصبح العنصر barracuda عند الفهرس 1، والعنصر cod عند الفهرس 2، والعنصر flounder - الأخير - عند الفهرس 5.

سنحضر الآن سمكة من نوع damselfish إلى الحوض، ونرغب في الحفاظ على الترتيب الأبجدي لعناصر القائمة أعلاه، لذلك سنضع هذا العنصر عند الفهرس 3: fish.insert(3,'damselfish').

ج. التابع `list.extend()`

إذا أردت أن توسّع قائمة بعناصر قائمة أخرى، فيمكنك استخدام التابع list.extend(L)، والذي يأخذ قائمة (المعامل L) ويضيف عناصرها إلى القائمة list.

سنضع في الحوض أربعة أسماك جديدة. أنواع هذه الأسماك مجموعة معًا في القائمة more_fish:

```
more_fish = ['goby','herring','ide','kissing gourami']
```

سنضيف الآن عناصر القائمة more_fish إلى قائمة الأسماك، ونطبع القائمة لنتأكد من أنَّ عناصر القائمة الثانية قد أضيفت إليها:

```
fish.extend(more_fish)
```

```
print(fish)
```

ستطبع بايثون القائمة التالية:

```
['anchovy', 'barracuda', 'cod', 'devil ray', 'eel', 'flounder',
'goby', 'herring', 'ide', 'kissing gourami']
```

في هذه المرحلة، صارت القائمة `fish` تتألف من 10 عناصر.

د. التابع `list.remove()`

لإزالة عنصر من قائمة، استخدم التابع `list.remove(x)`، والذي يزيل أول عنصر من القائمة له القيمة المُمرَّرة x.

جاءت مجموعة من العلماء المحليين لزيارة الحوض، وسيجرون أبحاثًا عن النوع `kissing gourami`، وطلبوا استعارة السمكة `kissing gourami`، لذلك نود إزالة العنصر `kissing gourami` من القائمة لنعكس هذا التغيير:

```
fish.remove('kissing gourami')
print(fish)
```

والمخرجات ستكون:

```
['anchovy', 'barracuda', 'cod', 'devil ray', 'eel', 'flounder',
'goby', 'herring', 'ide']
```

بعد استخدام التابع `list.remove()`، لم يعد العنصر `kissing gourami` موجودًا في القائمة.

في حال اخترت عنصرًا x غير موجود في القائمة ومرَّرته إلى التابع `list.remove()`، فسيُطلق الخطأ التالي:

```
ValueError: list.remove(x): x not in list
```

لن يزيل التابع `list.remove()` إلا أوّل عنصر تساوي قيمته قيمة العنصر المُمرَّر إلى التابع، لذلك إن كانت لدينا سمكتان من النوع `kissing gourami` في الحوض، وأعرنا إحداهما فقط للعلماء، فإنَّ التعبير `fish.remove('kissing gourami')` لن يمحو إلا العنصر الأول المطابق فقط.

٥. التابع `list.pop()`

يعيد التابع `list.pop ([i])` العنصر الموجود عند الفهرس المحدد من القائمة، ثم يزيل ذلك العنصر. تشير الأقواس المربعة حول `i` إلى أنَّ هذا المعامل اختياري، لذا، إذا لم تحدد فهرسًا (كما في `fish.pop()`)، فسيُعاد العنصر الأخير ثم يُزال.

لقد أصبح حجم السمكة `devil ray` كبيرًا جدًا، ولم يعد الحوض يسعها، ولحسن الحظ أنَّ هناك حوض سمك في بلدة مجاورة يمكنه استيعابها. سنستخدم التابع `.pop()`، ونمرر إليه العدد 3، الذي يساوي فهرس العنصر `devil ray`، بقصد إزالته من القائمة. بعد إعادة العنصر، سنتأكد من أننا أزلنا العنصر الصحيح.

```
print(fish.pop(3)) # devil ray
print(fish)
# ['anchovy', 'barracuda', 'cod', 'eel', 'flounder', 'goby',
'herring', 'ide']
```

باستخدام التابع `.pop()` تمكّنا من إزالة السمكة `devil ray` من قائمة الأسماك. إذا لم نُمرِّر أيَّ معامل إلى هذا التابع، ونفَّذنا الاستدعاء `fish.pop()`، فسيُعاد العنصر الأخير `ide` ثم يُزَال من القائمة.

و. التابع ()list.index

يصعب في القوائم الكبيرة تحديد فهارس العناصر التي تحمل قيمة معينة. لأجل ذلك، يمكننا استخدام التابع list.index(x)، إذ يمثل الوسيط x قيمة العنصر المبحوث عنه، والذي نريد معرفة فهرسه. إذا كان هناك أكثر من عنصر واحد يحمل القيمة x، فسيُعَاد فهرس العنصر الأول.

```
print(fish)    # ['anchovy', 'barracuda', 'cod', 'eel',
'flounder', 'goby', 'herring', 'ide']

print(fish.index('herring'))    # 6
```

سوف يُطلَق خطأ في حال مرَّرنا قيمة غير موجودة في القائمة إلى التابع ()index..

ز. التابع ()list.copy

أحيانًا نرغب في تعديل عناصر قائمةٍ والتجريب عليها، مع الحفاظ على القائمة الأصلية دون تغيير؛ يمكننا في هذه الحالة استخدام التابع ()list.copy لإنشاء نسخة من القائمة الأصلية.

في المثال التالي، سنمرِّر القيمة المعادة من ()fish.copy إلى المتغير fish_2، ثم نطبع قيمة fish_2 للتأكد من أنَّها تحتوي على نفس عناصر القائمة fish.

```
fish_2 = fish.copy()
print(fish_2)
#  ['anchovy', 'barracuda', 'cod', 'eel', 'flounder', 'goby',
'herring', 'ide']
```

في هذه المرحلة، القائمتان fish و fish_2 متساويتان.

ح. التابع `list.reverse()`

يمكننا عكس ترتيب عناصر قائمة باستخدام التابع `list.reverse()`. في المثال التالي سنستخدم التابع `.reverse()` مع القائمة `fish` لعكس ترتيب عناصرها.

```
fish.reverse()
print(fish)    #  ['ide', 'herring', 'goby', 'flounder', 'eel', 'cod', 'barracuda', 'anchovy']
```

بعد استخدام التابع `.reverse()`، صارت القائمة تبدأ بالعنصر `ide`، والذي كان في نهاية القائمة من قبل، كما ستنتهي القائمة بالعنصر `anchovy`، والذي كان في بداية القائمة من قبل.

ط. التابع `list.count()`

يعيد التابع `list.count(x)` عدد مرات ظهور القيمة `x` في القائمة. هذا التابع مفيد في حال كنا نعمل على قائمة طويلة بها الكثير من القيم المتطابقة.

إذا كان حوض السمك كبيرًا، على سبيل المثال، وكانت عندنا عدة أسماك من النوع `neon tetra`، فيمكننا استخدام التابع `.count()` لتحديد العدد الإجمالي لأسماك هذا النوع.

في المثال التالي، سنحسب عدد مرات ظهور العنصر `goby`:

```
print(fish.count('goby'))      #   1
```

تظهر السلسلة النصية `goby` مرةً واحدةً فقط في القائمة، لذا سيُعيد التابع `.count()` العدد 1. يمكننا استخدام هذا التابع أيضًا مع قائمة مكوَّنة من أعداد صحيحة، ويوضح المثال التالي ذلك.

يتتبع المشرفون على الحوض أعمار الأسماك الموجودة فيه للتأكد من أنَّ وجباتها الغذائية مناسبة لأعمارها. هذه القائمة الثانية المُسماة `fish_ages` تتوافق مع أنواع السمك في

القائمة fish. نظرًا لأنّ الأسماك التي لا يتجاوز عمرها عامًا واحدًا لها احتياجات غذائية خاصة، فسنحسب عدد الأسماك التي عمرها عامًا واحدًا:

```
fish_ages = [1,2,4,3,2,1,1,2]
print(fish_ages.count(1))    #  3
```

يظهر العدد الصحيح 1 في القائمة fish_ages ثلاث مرات، لذلك يعيد التابع ()count. العدد 3.

ي. التابع ()list.sort

يُستخدم التابع ()list.sort لترتيب عناصر القائمة التي استُدعِي معها. سنستخدم قائمة الأعداد الصحيحة fish_ages لتجريب التابع ()sort.:

```
fish_ages.sort()
print(fish_ages)    #  [1, 1, 1, 2, 2, 2, 3, 4]
```

باستدعاء التابع ()sort. مع القائمة fish_ages، فستُعاد تلك القائمة مرتَّبةً.

ك. التابع ()list.clear

بعد الانتهاء من العمل على قائمة ما، يمكنك إزالة جميع العناصر الموجودة فيها باستخدام التابع ()list.clear.

قررت الحكومة المحلية الاستيلاء على حوض السمك الخاص بنا، وجعله مساحة عامة يستمتع بها سكان مدينتنا. نظرًا لأننا لم نعد نعمل على الحوض، فلم نعد بحاجة إلى الاحتفاظ بقائمة الأسماك، لذلك سنزيل عناصر القائمة fish:

```
fish.clear()
print(fish)    #  []
```

نرى في المخرجات أقواسًا معقوفة نتيجة استدعاء التابع ‎.clear()‎ على القائمة fish، وهذا تأكيد على أنَّ القائمة أصبحت خالية من جميع العناصر.

8. فهم كيفية استعمال List Comprehensions

توفر List Comprehensions طريقةً مختصرةً لإنشاء القوائم بناءً على قوائم موجودة مسبقًا. فيمكن عند استخدام list comprehensions بناء القوائم باستخدام أيِّ نوع من البيانات المتسلسلة التي يمكن الدوران على عناصرها عبر حلقات التكرار، بما في ذلك السلاسل النصية والصفوف. من ناحية التركيب اللغوي، تحتوي list comprehensions على عنصر يمكن المرور عليه ضمن تعبيرٍ متبوعٍ بحلقة for. ويمكن أن يُتبَع ما سبق بتعابير for أو if إضافية، لذا سيساعدك الفهم العميق لحلقات for والعبارات الشرطية في التعامل مع list comprehensions.

توفِّر list comprehensions طريقةً مختلفةً لإنشاء القوائم وغيرها من أنواع البيانات المتسلسلة. وعلى الرغم من إمكانية استخدام الطرائق الأخرى للدوران، مثل حلقات for، لإنشاء القوائم، لكن من المفضَّل استعمال list comprehensions لأنها تقلِّل عدد الأسطر الموجودة في برنامجك.

يمكن بناء list comprehensions في بايثون كالآتي:

```
list_variable = [x for x in iterable]
```

ستُسنَد القائمة، أو أي نوع من البيانات يمكن المرور على عناصره، إلى متغير. المتغيرات الإضافية -التي تُشير إلى عناصر موجودة ضمن نوع البيانات الذي يمكن المرور على عناصره-

تُبنى حول عبارة for. والكلمة المحجوزة in تستعمل بنفس استعمالها في حلقات for وذلك لمرور على عناصر iterable. لننظر إلى مثالٍ يُنشِئ قائمةً مبنيةً على سلسلةٍ نصية:

```
shark_letters = [letter for letter in 'shark']
print(shark_letters)
```

أسندنا في المثال السابق قائمةً جديدةً إلى المتغير shark_letters، واستعملنا المتغير letter للإشارة إلى العناصر الموجودة ضمن السلسلة النصية 'shark'. استعملنا بعد ذلك الدالة print() لكي نتأكد من القائمة الناتجة والمُسنَدة إلى المتغير shark_letters، وحصلنا على الناتج الآتي:

```
['s', 'h', 'a', 'r', 'k']
```

تتألف القائمة التي أنشأناها باستخدام list comprehensions من العناصر التي تكوِّن السلسلة النصية 'shark'، وهي كل حرف في الكلمة shark. يمكن إعادة كتابة تعابير list comprehensions بشكل حلقات for، لكن لاحظ أنَّك لا تستطيع إعادة كتابة كل حلقة for بصيغة list comprehensions. لنعد كتابة المثال السابق الذي أنشأنا فيه القائمة shark_letters باستخدام حلقة for، وهذا سيساعدنا في فهم كيف تعمل list comprehensions عملها:

```
shark_letters = []

for letter in 'shark':
    shark_letters.append(letter)

print(shark_letters)
```

عند إنشائنا للقائمة عبر استخدام الحلقة for، فيجب تهيئة المتغير الذي ستُسنِد العناصر

إليه كقائمة فارغة، وهذا ما فعلناه في أوّل سطر من الشيفرة السابقة. ثم بدأت حلقة for بالدوران على عناصر السلسلة النصية 'shark' مستعملةً المتغير letter للإشارة إلى قيمة العنصر الحالي، ومن ثم أضفنا كل عنصر في السلسلة النصية إلى القائمة ضمن حلقة for وذلك باستخدام الدالة list.append(x). الناتج من حلقة for السابقة يماثل ناتج list comprehension في المثال أعلاه:

```
['s', 'h', 'a', 'r', 'k']
```

يمكن إعادة كتابة List comprehensions كحلقات for، لكن بعض حلقات for يمكن إعادة كتابتها لتصبح List comprehensions لتقليل كمية الشيفرات المكتوبة.

ا. استخدام التعابير الشرطية مع List Comprehensions

يمكن استخدام التعابير الشرطية في list comprehension لتعديل القوائم أو أنواع البيانات المتسلسلة الأخرى عند إنشاء قوائم جديدة. لننظر إلى مثالٍ عن استخدام العبارة الشرطية if في تعبير list comprehension:

```
fish_tuple = ('blowfish', 'clownfish', 'catfish', 'octopus')

fish_list = [fish for fish in fish_tuple if fish != 'octopus']
print(fish_list)
```

استعملنا المتغير fish_tuple الذي من نوع البيانات tuple أساسًا للقائمة الجديدة التي سنُنشِئها التي تسمى fish_list. استعملنا for و in كما في القسم السابق، لكننا أضفنا هنا التعليمة الشرطية if. ستؤدي التعليمة الشرطية if إلى إضافة العناصر غير المساوية للسلسلة النصية 'octopus'، لذا ستحتوي القائمة الجديدة على العناصر الموجودة في بنية صف

(tuple) والتي لا تُطابِق الكلمة 'octopus'. عند تشغيل البرنامج السابق فسنلاحظ أنَّ القائمة fish_list تحتوي على نفس العناصر التي كانت موجودة في fish_tuple لكن مع حذف العنصر 'octopus':

```
['blowfish', 'clownfish', 'catfish']
```

أي أصبحت القائمة الجديدة تحتوي على بنية صف أصلية لكن ما عدا السلسلة النصية التي استثنيناها عبر التعبير الشرطي. سنُنشِئ مثالًا آخر يستعمل المعاملات الرياضية والأرقام الصحيحة والدالة range():

```
number_list = [x ** 2 for x in range(10) if x % 2 == 0]
print(number_list)
```

القائمة التي ستُنشَأ باسم number_list ستحتوي على مربع جميع القيم الموجودة من المجال 0 إلى 9 لكن إذا كان الرقم قابلًا للقسمة على 2. وستبدو المخرجات الآتية:

```
[0, 4, 16, 36, 64]
```

دعنا نُفصِّل ما الذي يفعله تعبير list comprehension السابق، ودعنا نفكِّر بالذي سيظهر إذا استعملنا التعبير x for x in range(10) فقط. يجب أن يبدو برنامجنا الصغير كالآتي:

```
number_list = [x for x in range(10)]
print(number_list)
```

الناتج:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

لنضف العبارة الشرطية الآن:

```
number_list = [x for x in range(10) if x % 2 == 0]
print(number_list)
```

الناتج:

```
[0, 2, 4, 6, 8]
```

أدّت التعليمة الشرطية `if` إلى قبول العناصر القابلة للقسمة على 2 فقط وإضافتها إلى القائمة، مما يؤدي إلى حذف جميع الأرقام الفردية. يمكننا الآن استخدام معامل رياضي لتربيع قيمة المتغير `x`:

```
number_list = [x ** 2 for x in range(10) if x % 2 == 0]
print(number_list)
```

أي ستُربَّع قيم القائمة السابقة `[0, 2, 4, 6, 8]` وسيُخرَج الناتج الآتي:

```
[0, 4, 16, 36, 64]
```

يمكننا أيضًا استعمال ما يشبه عبارات `if` المتشعبة في تعابير list comprehension:

```
number_list = [x for x in range(100) if x % 3 == 0 if x % 5 ==
0]
print(number_list)
```

سيتم التحقُّق أولًا أنَّ المتغير `x` قابل للقسمة على الرقم 3، ثم سنتحقق إن كان المتغير `x` قابل للقسمة على الرقم 5، وإذا حقَّق المتغير `x` الشرطين السابقين فسيُضاف إلى القائمة، وسيُظهَر في الناتج:

```
[0, 15, 30, 45, 60, 75, 90]
```

يمكن استخدام تعليمة if الشرطية لتحديد ما هي العناصر التي نريد إضافتها إلى القائمة الجديدة.

ب. حلقات التكرار المتشعبة في تعابير List Comprehension

يمكن استعمال حلقات التكرار المتشعبة لإجراء عدّة عمليات دوران متداخلة في برامجنا.

سننظر في هذا القسم إلى حلقة for متشعبة وسنحاول تحويلها إلى تعبير list comprehension. ستُنشِئ هذه الشيفرة قائمةً جديدةً بالدوران على قائمتين وبإجراء عمليات رياضية عليها:

```
my_list = []

for x in [20, 40, 60]:
   for y in [2, 4, 6]:
     my_list.append(x * y)

print(my_list)
```

سنحصل على الناتج الآتي عند تشغيل البرنامج:

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```

تضرب الشيفرة السابقة العناصر الموجودة في أوَّل قائمة بالعناصر الموجودة في ثاني قائمة في كل دورة. لتحويل ما سبق إلى تعبير list comprehension، وذلك باختصار السطرين الموجودين في الشيفرة السابقة وتحويلهما إلى سطرٍ وحيدٍ، الذي يبدأ بإجراء العملية x*y، ثم ستلي هذه العملية حلقة for الخارجية، ثم يليها حلقة for الداخلية؛ وسنضيف تعبير print() للتأكد أنَّ ناتج القائمة الجديدة يُطابِق ناتج البرنامج الذي فيه حلقتين متداخلتين:

```
my_list = [x * y for x in [20, 40, 60] for y in [2, 4, 6]]
print(my_list)
```

الناتج:

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```

أدى استعمال تعبير list comprehension في المثال السابق إلى تبسيط حلقتَي for لتصبحا سطرًا وحيدًا، لكن مع إنشاء نفس القائمة والتي ستُسنَد إلى المتغير my_list. توفِّر لنا تعابير list comprehension طريقةً بسيطةً لإنشاء القوائم، مما يسمح لنا باختصار عدّة أسطر إلى سطرٍ وحيد. لكن من المهم أن تبقي في ذهنك أنَّ سهولة قراءة الشيفرة لها الأولوية دومًا، لذا إذا أصبحتَ تعابير list comprehension طويلةً جدًا ومعقَّدة، فمن الأفضل حينها تحويلها إلى حلقات تكرار عادية.

9. خلاصة الفصل

القوائم هي نوع بيانات مرن يمكن تعديله بسهولة خلال أطوار البرنامج. غطينا في هذا الفصل الميزات والخصائص الأساسية لقوائم، بما في ذلك الفهرسة والاقتطاع والتعديل والضَّم. لمَّا كانت القوائم تسلسلات قابلة للتغيير (mutable)، فإنَّها هياكلُ بيانات مرنة ومفيدة للغاية. كما تتيح لنا توابع القوائم إجراء العديد من العمليات على القوائم بسهولة، إذ يمكننا استخدام التوابع لتعديل القوائم وترتيبها ومعالجتها بفعالية.

تسمح تعابير list comprehension لنا بتحويل قائمة أو أي نوع من البيانات المتسلسلة إلى سلسلةٍ جديدة، ولها شكلٌ بسيطٌ يُقلِّل عدد الأسطر التي نكتبها. تتبع تعابير list comprehension شكلًا رياضيًا معيّنًا، لذا قد يجدها المبرمجون أولو الخلفية الرياضية سهلة الفهم. وصحيحٌ أنَّ

تعـابير list comprehension تختصـر الشيفرة. لكن من المهم جعـل سـهولة قـراءة الشـيفرة من أولوياتنا، وحاول تجنُّب الأسطر الطويلة لتسهيل قراءة الشيفرة.

12

النوع Tuple: فهم الصفوف

يبدو نوع البيانات tuple (صف) في بايثون كما يلي:

```
coral = ('blue coral', 'staghorn coral', 'pillar coral',
'elkhorn coral')
```

النوع tuple (صف وتُجمَع إلى صفوف) هي بنية بيانات تُمثِّل سلسلة مرتبة من العناصر غير القابلة للتبديل، وبالتالي لا يمكن تعديل القيم الموجودة فيها. يستعمل نوع البيانات tuple لتجميع البيانات، فكل عنصر أو قيمة داخل الصف تُشكِّل جزءًا منه. توضع القيم داخل الصف بين قوسين ( ) ويُفصَل بينها بفاصلة أجنبية , وتبدو القيم الفارغة كما يلي `coral = ()`، لكن إذا احتوى الصف على قيم -حتى لو كانت قيمةً واحدةً فقط- فيجب وضع فاصلة فيه مثل `coral = ('blue coral',)`. إذا استخدمنا الدالة `print()` على النوع tuple، فسنحصل على الناتج الآتي الذي يُبيّن أنَّ القيمة الناتجة ستوضع بين قوسين:

```
print(coral)
('blue coral', 'staghorn coral', 'pillar coral', 'elkhorn
coral')
```

عند التفكير بالنوع tuple وغيره من بنى البيانات التي تُعدُّ من أنواع «التجميعات» (collections)، فمن المفيد أن تضع ببالك مختلف التجميعات الموجودة في حاسوبك: تشكيلة الملفات الموجودة عندك، وقوائم التشغيل للموسيقى، والمفضلة الموجودة في متصفحك، ورسائل بريدك الإلكتروني، ومجموعة مقاطع الفيديو التي تستطيع الوصول إليها من التلفاز، والكثير. نوع tuple (صف) شبيه بالنوع list (قائمة)، لكن القيم الموجودة فيه لا يمكن تعديلها، وبسبب ذلك، فأنت تخبر الآخرين أنَّك لا تريد إجراء أيَّة تعديلات على هذه السلسلة من القيم عندما تستعمل النوع tuple في شيفرتك. إضافةً إلى ما سبق، ولعدم القدرة على تعديل القيم،

فسيكون أداء برنامجك أفضل، حيث ستُنفَّذ الشيفرة بشكل أسرع إذا استعملتَ الصفوف بدلًا من القوائم (lists).

1. فهرسة الصفوف

يمكن الوصول إلى كل عنصر من عناصر الصف بمفرده لأنَّه سلسلة مرتبة من العناصر، وذلك عبر الفهرسة. وكل عنصر يرتبط برقم فهرس، الذي هو عدد صحيح يبدأ من الفهرس 0. ستبدو الفهارس من مثال coral السابق والقيم المرتبطة بها كالآتي:

blue coral	staghorn coral	pillar coral	elkhorn coral
0	1	2	3

العنصر الأول الذي يُمثِّل السلسلة النصية 'blue coral' فهرسه 0، وتنتهي القائمة بالفهرس رقم 3 المرتبط بالقيمة 'elkhorn coral'. ولأنَّ كل عنصر من عناصر الصف له رقم فهرس مرتبط به، فستتمكن من الوصول إلى عناصره فرادى. يمكننا الآن الوصول إلى عنصر معيّن في الصف عبر استخدام رقم الفهرس المرتبط به.

```
print(coral[2])
pillar coral
```

تتراوح قيم الفهارس في المتغير coral من 0 إلى 3 كما هو ظاهر في الجدول السابق، لذا يمكننا استدعاء العناصر الموجودة فيه فرادى كما يلي:

```
coral[0]
coral[1]
coral[2]
```

```
coral[3]
```

إذا حاولنا استدعاء المتغير coral مع رقم فهرس أكبر من 3، فستظهر رسالة خطأ تشير إلى أنَّ الفهرس خارج المجال:

```
print(coral[22])
# IndexError: tuple index out of range
```

إضافةً إلى أرقام الفهارس الموجبة، يمكننا أيضًا الوصول إلى الفهارس باستخدام رقم فهرس سالب، وذلك بالعد بدءًا من نهاية قائمة العناصر وسيرتبط آخر عنصر بالفهرس ‎-1، وهذا مفيدٌ جدًا إذا كان لديك متغير من النوع tuple وكان يحتوي عناصر كثيرة وأردتَ الوصول إلى أحد عناصره انطلاقًا من النهاية. ففي مثالنا السابق عن coral، إذا أردنا استخدام الفهارس السالبة فالناتج كالآتي:

elkhorn coral	pillar coral	staghorn coral	blue coral
-1	-2	-3	-4

إذا أردنا طباعة العنصر 'blue coral' باستخدام الفهارس السالبة، فستبدو التعليمة كما يلي:

```
print(coral[-4])
blue coral
```

يمكننا إضافة العناصر النصية الموجودة في الصف إلى السلاسل النصية الأخرى باستخدام العامل +:

```
print('This reef is made up of ' + coral[1])
# This reef is made up of staghorn coral
```

استطعنا في المثال السابق إضافة عنصر موجود في الفهرس 1 مع السلسلة النصية `'This reef is made up of '`، ويمكننا أيضًا استخدام العامل `+` لإضافة بنيتَي صف معًا.

2. تقطيع قيم صف

يمكننا استخدام الفهارس للوصول إلى عدّة عناصر من صف، أما التقطيع فيسمح لنا بالوصول إلى عدّة قيم عبر إنشاء مجال من أرقام الفهارس المفصولة بنقطتين رأسيتين `[x:y]`. لنقل أنّنا نريد عرض العناصر الموجودة في وسط المتغير `coral`، يمكننا فعل ذلك بإنشاء قطعة جديدة:

```
print(coral[1:3])
('staghorn coral', 'pillar coral')
```

عند إنشاء قطعة جديدة -كما في المثال السابق- فيمثّل أوّل رقم مكان بدأ القطعة (متضمنةً هذا الفهرس)، ورقم الفهرس الثاني هو مكان نهاية القطعة (دون تضمين هذا الفهرس بالقطعة)، وهذا هو السبب وراء عرض المثال السابق للقيم المرتبطة بالعناصر الموجودة في الفهرسين 1 و 2. إذا أردتَ تضمين إحدى نهايتَي القائمة، فيمكنك حذف أحد الأرقام في التعبير `tuple[x:y]`، فمثلًا، لنقل أننا نريد عرض أوّل ثلاثة عناصر من `coral`، والتي هي `'blue coral'` و `'staghorn coral'` و `'pillar coral'`، فيمكننا فعل ذلك كالآتي:

```
print(coral[:3])
('blue coral', 'staghorn coral', 'pillar coral')
```

المثال السابق عرض العناصر من بداية القائمة وتوقف قبل العنصر ذي الفهرس 3. لتضمين

كل العناصر الموجودة في نهاية الصف، فيمكننا عكس التعبير السابق:

```
print(coral[1:])
# ('staghorn coral', 'pillar coral', 'elkhorn coral')
```

يمكننا استخدام الفهارس السالبة أيضًا عند التقطيع، كما فعلنا مع أرقام الفهارس الموجبة:

```
print(coral[-3:-1])
print(coral[-2:])
# ('staghorn coral', 'pillar coral')
# ('pillar coral', 'elkhorn coral')
```

هنالك معاملٌ إضافيٌ يمكننا استعماله ويسمى «الخطوة»، ويُشير إلى عدد العناصر التي يجب تجاوزها بعد الحصول على أوّل عنصر من القائمة. حذفنا في جميع أمثلتنا السابقة معامل الخطوة، إذ القيمة الافتراضية له في بايثون هي 1، لذا سنحصل على جميع العناصر الموجودة بين الفهرسَين المذكورين. شكل هذا التعبير العام هو `tuple[x:y:z]`، إذ يُشير المعامل `z` إلى الخطوة. لنُنشِئ قائمةً أكبر، ثم نقسّمها، ونعطيها القيمة 2 للخطوة:

```
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
print(numbers[1:11:2])
# (1, 3, 5, 7, 9)
```

ستطبع التعليمة `numbers[1:11:2]` القيم الموجودة بين رقمين الفهرسين 1 (بما في ذلك العنصر المرتبط بالفهرس 1) و 11 (دون تضمين ذلك العنصر)، ومن ثم ستخبر قيمةُ الخطوة 2 البرنامجَ أنَّ يتخطى عنصرًا بين كل عنصرين. يمكننا حذف أوَّل معاملين واستخدام معامل الخطوة بمفرده بتعبيرٍ برمجي من الشكل `tuple[::z]`:

```
print(numbers[::3])
# (0, 3, 6, 9, 12)
```

طبعنا في المثال السابق عناصر numbers بعد ضبط قيمة الخطوة إلى 3، وبالتالي سيتم تخطي عنصرين.

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

تقطيع الصفوف باستخدام أرقام الفهارس الموجبة والسالبة واستعمال معامل الخطوة يسمح لنا بالتحكم بالناتج الذي نريد عرضه.

3. إضافة بنى صف إلى بعضها

يمكن أن نُضيف بنى صف إلى بعضها أو أن «نضربها» (multiply)، تتم عملية الإضافة باستخدام المعامل +، أما عملية الضرب فباستخدام المعامل *. يمكن أن يُستخدَم المعامل + لإضافة بنيتَي صف أو أكثر إلى بعضها بعضًا. يمكننا إسناد القيم الموجودة في بنيتَي صف إلى بنية جديدة:

```
coral = ('blue coral', 'staghorn coral', 'pillar coral',
'elkhorn coral')
kelp = ('wakame', 'alaria', 'deep-sea tangle', 'macrocystis')

coral_kelp = (coral + kelp)

print(coral_kelp)
```

الناتج:

```
('blue coral', 'staghorn coral', 'pillar coral', 'elkhorn
coral', 'wakame', 'alaria', 'deep-sea tangle', 'macrocystis')
```

وصحيحٌ أنَّ المعامل + يمكنه إضافة بنى صف إلى بعضها، لكن يمكن أن يستعمل لإنشاء بنية صف جديدة ناتجة عن جمع بنى أخرى، لكن لا يمكنه تعديل بنية صف موجودة مسبقًا. أما العامل * فيمكن استخدامه لضرب بنى صف، فربما تريد إنشاء نسخ من الملفات الموجودة في أحد المجلدات إلى الخادوم أو مشاركة قائمة بالمقطوعات الموسيقية التي تحبها مع أصدقائك، ففي هذه الحالات سترغب بمضاعفة مجموعات من البيانات (أو «ضربها»). لنضرب البنية coral بالرقم 2 والبنية kelp بالرقم 3، ثم نسندها إلى بنى صف جديدة:

```
multiplied_coral = coral * 2
multiplied_kelp = kelp * 3

print(multiplied_coral)
print(multiplied_kelp)
```

الناتج:

```
('blue coral', 'staghorn coral', 'pillar coral', 'elkhorn
coral', 'blue coral', 'staghorn coral', 'pillar coral',
'elkhorn coral')
('wakame', 'alaria', 'deep-sea tangle', 'macrocystis',
'wakame', 'alaria', 'deep-sea tangle', 'macrocystis', 'wakame',
'alaria', 'deep-sea tangle', 'macrocystis')
```

يمكننا باستخدام العامل * أن نُكرِّر (أو نُضاعِف) بنى صف بأي عدد من المرات نشاء، مما سيُنشِئ بنى صف جديدة اعتمادًا على محتوى البنى الأصلية. خلاصة ما سبق هي أنَّ بنى الصف يمكن إضافتها إلى بعضها أو ضربها لتشكيل بنى صف جديدة عبر استخدام العاملَين + و *.

4. دوال التعامل مع الصفوف

هنالك دوال مُضمَّنة في لغة بايثون للتعامل مع بنى النوع tuple، لننظر إلى بعضها.

أ. ()len

وكما في السلاسل النصية والقوائم، يمكننا حساب طول (أو عدد عناصر) بنية صف باستخدام الدالة ()len إذ نُمرِّر إليها بنية صف (معامل)، كما يلي:

```
len(coral)
```

هذه الدالة مفيدة إذا أردنا أن نَضمَن أنَّ لبنية صف عدد عناصر معيَّن، فمثلًا يمكننا الاستفادة من ذلك بموازنة بنيتين مع بعضهما. إذا أردنا طباعة عدد عناصر kelp و numbers، فسيظهر الناتج الآتي:

```
print(len(kelp))
print(len(numbers))
```

الناتج:

```
4
13
```

يشير الناتج أعلاه إلى أنَّ للبنية kelp أربعة عناصر:

```
kelp = ('wakame', 'alaria', 'deep-sea tangle', 'macrocystis')
```

أمَّا البنية numbers فتملك ثلاثة عشر عنصرًا:

```
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
```

وصحيحٌ أنَّ هذه الأمثلة عناصرها قليلة نسبيًا، إلا أنَّ الدالة `len()` تستطيع أن تخبرنا بعدد عناصر بنى `tuple` الكبيرة.

ب. الدالتان `max()` و `min()`

عندما نتعامل مع بنى صف مكوَّنة من عناصر رقمية (بما فيها الأعداد الصحيحة والأرقام ذات الفاصلة العشرية)، فيمكننا استخدام الدالتين `max()` و `min()` للعثور على أكبر وأصغر قيمة موجودة في بنية صف معيَّنة. تسمح لنا هاتان الدالتان باستخراج معلومات تخص البيانات القابلة للإحصاء، مثل نتائج الامتحانات أو درجات الحرارة أو أسعار المنتجات ...إلخ. لننظر إلى بنية صف مكونة من أعداد عشرية:

```
more_numbers = (11.13, 34.87, 95.59, 82.49, 42.73, 11.12,
95.57)
```

للحصول على القيمة العظمى من بين القيم الآتية فعلينا تمرير بنية صف إلى الدالة `max()` كما في `max(more_numbers)`، وسنستخدم الدالة `print()` لعرض الناتج:

```
print(max(more_numbers))
# 95.59
```

أعادت الدالة `max()` أعلى قيمة في بنية `more_numbers`. وبشكلٍ شبيهٍ بما سبق نستخدم الدالة `min()`:

```
print(min(more_numbers))
# 11.12
```

أُعيدَ هنا أصغر رقم عشري موجودة في البنية. يمكن الاستفادة من الدالتين `max()` و `min()` كثيرًا للتعامل مع بنى `tuple` التي تحتوي الكثير من القيم.

5. كيف تختلف بنى الصفوف عن القوائم

الفرق الرئيسي بين النوع tuple والنوع list هو عدم القدرة على تعديل العناصر، وهذا يعني أنَّنا لا نستطيع إضافة أو حذف أو استبدال العناصر داخل بنية tuple. لكن يمكننا إضافة بنيتَي tuple أو أكثر إلى بعضها بعضًا لتشكيل بنية جديدة كما رأينا في أحد الأقسام السابقة.

لتكن لدينا البنية coral الآتية:

```
coral = ('blue coral', 'staghorn coral', 'pillar coral',
'elkhorn coral')
```

لنقل أننا نريد استبدال العنصر 'blue coral' ووضع العنصر 'black coral' بدلًا منه. فلو حاولنا تغيير بنية صف بنفس الطريقة التي نُعدِّل فيها القوائم بكتابة:

```
coral[0] = 'black coral'
```

فستظهر رسالة خطأ كالآتية:

```
TypeError: 'tuple' object does not support item assignment
```

وذلك بسبب عدم إمكانية تعديل بنى الصفوف. إذا أنشأنا بنية صف ثم قررنا أنَّ ما نحتاج له هو بنية قائمة، فيمكننا تحويلها إلى قائمة list، وذلك بالدالة list():

```
list(coral)
```

أصبحت بنية coral قائمةً الآن:

```
coral = ['blue coral', 'staghorn coral', 'pillar coral']
```

يمكننا أن نلاحظ أنَّ بنية الصف tuple تحوَّلت إلى قائمة list لأنَّ الأقواس المحيطة بالقيم أصبحت مربعة الشكل.

وبشكلٍ شبيهٍ بما سبق، نستطيع تحويل القوائم من النوع `list` إلى `tuple` باستخدام الدالة `tuple()`.

6. خلاصة الفصل

نوع البيانات `tuple` (الصفوف) هو مجموعةٌ من البيانات المتسلسلة التي لا يمكن تعديلها، ويوفِّر تحسينًا في أداء برامجك لأنه أسرع معالجةً من القوائم في بايثون. وعندما يراجع الآخرون شيفرتك فسيعلمون من استخدامك لبنى `tuple` أنك لا تريد تعديل هذه القيم. شرحنا في هذا الفصل الميزات الأساسية لبنى `tuple` بما في ذلك الفهارس وتقطيعها وتجميعها، وعرضنا بعض الدوال المُضمَّنة المتوافرة لهذا النوع من البيانات.

13

النوع Dictionary: فهم القواميس

النوع `dictionary` (القاموس) هو نوع مُضمَّن في بايثون. تربط القواميس مفاتيح بقيم على هيئة أزواج، وهذه الأزواج مفيدة لتخزين البيانات في بايثون.

تستخدم القواميس عادةً لتخزين البيانات المترابطة، مثل المعلومات المرتبطة برقم تعريف، أو ملفات تعريف المستخدم، وتُنشأ باستخدام الأقواس المعقوصة `{}`.

تبدو القواميس على الشكل التالي:

```
sammy = {'username': 'sammy-shark', 'online': True,
'followers': 987}
```

بالإضافة إلى القوسين المعقوصين، لاحظ وجود النقطتين الرأسيتين (`:`) في القاموس.

الكلمات الموجودة على يسار النقطتين الرأسيتين هي المفاتيح (keys) التي قد تكون أيَّ نوع بيانات غير قابل للتغيير. المفاتيح في القاموس أعلاه هي:

- `username`
- `online`
- `followers`

المفاتيح في المثال أعلاه عبارة عن سلاسل نصية.

تمثِّل الكلمات الموجودة على يمين النقطتين «القيم» (values). يمكن أن تتألف القيم من أي نوع من البيانات. القيم في القاموس أعلاه هي:

- `sammy-shark`
- `True`
- `87`

قيم القاموس أعلاه هي إمّا سلاسل نصية أو قيم منطقية أو أعداد صحيحة. سنطبع الآن القاموس `sammy`:

```
print(sammy)
```

الناتج:

```
{'username': 'sammy-shark', 'followers': 987, 'online': True}
```

نلاحظ بالنظر إلى المخرجات تغير ترتيب الأزواج قيمة-مفتاح (key-value). في الإصدار بايثون 3.5 وما قبله، كانت القواميس غير مرتبة. لكن ابتداءً من بايثون 3.6، صارت القواميس مرتبةً. بغض النظر عما إذا كان القاموس مرتبًا أم لا، ستظل الأزواج قيمة-مفتاح كما هي، وهذا سيمكّنك من الوصول إلى البيانات بناء على ترابطاتها.

1. الوصول إلى عناصر قاموس

يمكننا الوصول إلى قيم محدَّدة في القاموس بالرجوع إلى المفاتيح المرتبطة بها ويمكن أيضًا الاستعانة ببعض التوابع الجاهزة للوصول إلى القيم أو المفاتيح أو كليهما.

ا. الوصول إلى عناصر القاموس باستخدام المفاتيح

إذا أردنا الحصول على اسم المستخدم في `Sammy`، فيمكننا ذلك عن طريق استدعاء `sammy['username']`. هذا مثال على ذلك:

```
print(sammy['username'])    #  sammy-shark
```

تتصرف القواميس مثل قواعد البيانات، فهي بدلًا من فهرسة العناصر بأعداد صحيحة، كما هو الحال في القوائم، فإنّها تُفهرس العناصر (أو قيم القاموس) بمفاتيح، ويمكنك عبر تلك المفاتيح الحصول على القيم المقابلة لها.

باستدعاء المفتاح username، سنحصل على القيمة المرتبطة به، وهي sammy-shark. وبالمِثل، يمكن استدعاء القيم الأخرى في القاموس sammy باستخدام نفس الصياغة:

```
sammy['followers']  # 987

sammy['online']    # True
```

ب. استخدام التوابع للوصول إلى العناصر

بالإضافة إلى استخدام المفاتيح للوصول إلى القيم، يمكننا أيضًا استخدام بعض التوابع المُضمّنة، مثل:

- dict.keys()‎: الحصول على المفاتيح
- dict.values()‎: الحصول على القيم
- dict.items()‎: الحصول على العناصر على هيئة قائمة من أزواج (key, value)

لإعادة المفاتيح، نستخدم التابع dict.keys()‎، كما يوضح المثال التالي:

```
print(sammy.keys())
#  dict_keys(['followers', 'username', 'online'])
```

تلقينا في المخرجات كائنَ عرض تكراري (iterable view object) من الصنف dict_keys يحوي المفاتيح ثم طُبِعت المفاتيح على هيئة قائمة.

يمكن استخدام هذا التابع للاستعلام من القواميس. على سبيل المثال، يمكننا البحث عن المفاتيح المشتركة بين قاموسين:

```
sammy = {'username': 'sammy-shark', 'online': True,
'followers': 987}
jesse = {'username': 'JOctopus', 'online': False, 'points':
723}

for common_key in sammy.keys() & jesse.keys():
    print(sammy[common_key], jesse[common_key])
```

يحوي القاموسان sammy و jesse معلومات تعريف المستخدم. كما أنَّ لهما مفاتيح مختلفة، لأنَّ لدى Sammy ملف تعريف اجتماعي يضم مفتاحًا followers يمثل المتابعين على الشبكة الاجتماعية، أما Jesse فلها ملف تعريف للألعاب يضم مفتاحًا points يمثل النقاط. كلا القاموسين يشتركان في المفتاحين username و online، ويمكن العثور عليهما عند تنفيذ هذا البُريمج:

```
sammy-shark JOctopus
True False
```

يمكننا بالتأكيد تحسين البرنامج لتسهيل قراءة المخرجات، ولكنَّ الغرض هنا هو توضيح إمكانية استخدام dict.keys() لرصد المفاتيح المشتركة بين عدَّة قواميس. هذا مفيد بشكل خاص عند العمل على القواميس الكبيرة.

وبالمثل، يمكننا استخدام التابع dict.values() للاستعلام عن القيم الموجودة في القاموس sammy على النحو التالي:

```
sammy = {'username': 'sammy-shark', 'online': True,
'followers': 987}

print(sammy.values())  #  dict_values([True, 'sammy-shark',
987])
```

يُعيد كلا التابعين `values()` و `keys()` قوائم غير مرتبة تضم مفاتيح وقيم القاموس `sammy` على هيئة كائِني عرضٍ من الصنف `dict_values` و `dict_keys` على التوالي.

إن أردت الحصول على الأزواج الموجودة في القاموس، فاستخدم التابع `items()`:

```
print(sammy.items())
```

المخرجات ستكون:

```
dict_items([('online', True), ('username', 'sammy-shark'),
('followers', 987)])
```

ستكون النتيجة المعادة على هيئة قائمة مكونة من أزواج `(key,  value)` من الصنف `dict_items`.

يمكننا التكرار (iterate) على القائمة المُعادة باستخدام الحلقة `for`. على سبيل المثال، يمكننا طباعة جميع مفاتيح وقيم القاموس المحدد، ثم جعلها أكثر مقروئية عبر إضافة سلسلة نصية توضيحية:

```
for key, value in sammy.items():
    print(key, 'is the key for the value', value)
```

وسينتج لنا:

```
online is the key for the value True
followers is the key for the value 987
username is the key for the value sammy-shark
```

كرَّرت الحلقة for على العناصر الموجودة في القاموس sammy، وطبعت المفاتيح والقيم سطرًا سطرًا، مع إضافة معلومات توضيحية.

2. تعديل القواميس

القواميس هي هياكل بيانات قابلة للتغيير (mutable)، أي يمكن تعديلها. في هذا القسم، سنتعلم كيفية إضافة عناصر إلى قاموس، وكيفية حذفها.

ا. إضافة وتغيير عناصر القاموس

يمكنك إضافة أزواج قيمة-مفتاح إلى قاموس دون استخدام توابع أو دوال باستخدام الصياغة التالية:

```
dict[key] = value
```

في المثال التالي، سنضيف زوجًا مفتاح-قيمة إلى قاموس يُسمى usernames:

```
usernames = {'Sammy': 'sammy-shark', 'Jamie': 'mantisshrimp54'}

usernames['Drew'] = 'squidly'

print(usernames)    #  {'Drew': 'squidly', 'Sammy': 'sammy-shark', 'Jamie': 'mantisshrimp54'}
```

لاحظ أنَّ القاموس قد تم تحديثه بالزوج 'Drew': 'squidly'.

نظرًا لأنَّ القواميس غير مرتبة، فيمكن أن يظهر الزوج المُضاف في أيِّ مكان في مخرجات القاموس. إذا استخدمنا القاموس usernames لاحقًا، فسيظهر فيه الزوج المضاف حديثًا.

يمكن استخدام هذه الصياغة لتعديل القيمة المرتبطة بمفتاح معيَّن. في هذه الحالة، سنشير إلى مفتاح موجود سلفًا، ونمرِّر قيمة مختلفة إليه.

سنعرِّف في المثال التالي قاموسًا باسم drew يمثِّل البيانات الخاصة بأحد المستخدمين على بعض الشبكات الاجتماعية. حصل هذا المستخدم على عدد من المتابعين الإضافيين اليوم، لذلك سنحدّث القيمة المرتبطة بالمفتاح followers ثم نستخدم التابع print() للتحقق من أنّ القاموس قد عُدِّل.

```
drew = {'username': 'squidly', 'online': True, 'followers': 
305}

drew['followers'] = 342

print(drew)
#  {'username': 'squidly', 'followers': 342, 'online': True}
```

في المخرجات نرى أنّ عدد المتابعين قد قفز من 305 إلى 342.

يمكننا استخدام هذه الطريقة لإضافة أزواج قيمة-مفتاح إلى القواميس عبر مدخلات المستخدم. سنكتب بريمجًا سريعًا، usernames.py، يعمل من سطر الأوامر ويسمح للمستخدم بإضافة الأسماء وأسماء المستخدمين المرتبطة بها:

```
# تعريف القاموس الأصلي
usernames = {'Sammy': 'sammy-shark', 'Jamie': 'mantisshrimp54'}
# إعداد الحلقة التكرارية while
while True:
    # اطلب من المستخدم إدخال اسم
    print('Enter a name:')

    # تعيين المدخلات إلى المتغير name
   name = input()

    # تحقق مما إذا كان الاسم موجودًا في القاموس ثم اطبع الرد
    if name in usernames:
         print(usernames[name] + ' is the username of ' + name)

    # إذا لم يكن الاسم في القاموس
    else:

        # اطبع الرد
        print('I don\'t have ' + name + '\'s username, what is 
it?')

        # خذ اسم مستخدم جديد لربطه بذلك الاسم
        username = input()

        # عين قيمة اسم المستخدم إلى المفتاح name
        usernames[name] = username

        # اطبع ردًا يبيّن أنّ البيانات قد حُدّثت
        print('Data updated.')
```

سننفِّذ البرنامج من سطر الأوامر:

```
python usernames.py
```

عندما ننفِّذ البرنامج، سنحصل على مخرجات مشابهة لما يلي:

```
Enter a name:
Sammy
sammy-shark is the username of Sammy
Enter a name:
Jesse
I don't have Jesse's username, what is it?
JOctopus
Data updated.
Enter a name:
```

عند الانتهاء من اختبار البرنامج، اضغط على CTRL + C للخروج من البرنامج. يمكنك تخصيص حرف لإنهاء البرنامج (مثل الحرف q)، وجعل البرنامج يُنصت له عبر التعليمات الشرطية.

يوضح هذا المثال كيف يمكنك تعديل القواميس بشكل تفاعلي. في هذا البرنامج، بمجرد خروجك باستخدام CTRL + C، ستفقد جميع بياناتك، إلا إن خزّنت البيانات في ملف.

يمكننا أيضًا إضافة عناصر إلى القواميس وتعديلها باستخدام التابع `dict.update()`. هذا التابع مختلف عن التابع `append()` الذي يُستخدم مع القوائم.

سنضيف المفتاح `followers` في القاموس `jesse` أدناه، ونَمنحه قيمة عددية صحيحة بواسطة التابع `jesse.update()`. بعد ذلك، سنطبع القاموس المُحدَّث.

```
jesse = {'username': 'JOctopus', 'online': False, 'points':
723}

jesse.update({'followers': 481})

print(jesse)    #  {'followers': 481, 'username': 'JOctopus',
```

```
'points': 723, 'online': False}
```

نتبيَّن من المخرجات أنَّنا نجحنا في إضافة الزوج `followers: 481` إلى القاموس `jesse`.

يمكننا أيضًا استخدام التابع `dict.update()` لتعديل زوج قيمة-مفتاح موجود سلفًا عن طريق استبدال قيمة مفتاح معيَّن.

سنغيِّر القيمة المرتبطة بالمفتاح `online` في القاموس `Sammy` من `True` إلى `False`:

```
sammy = {'username': 'sammy-shark', 'online': True,
'followers': 987}

sammy.update({'online': False})

print(sammy)    #  {'username': 'sammy-shark', 'followers': 987,
'online': False}
```

يغيّر السطر `sammy.update({'online': False})` القيمة المرتبطة بالمفتاح `'online'` من `True` إلى `False`. عند استدعاء التابع `print()` على القاموس، يمكنك أن ترى في المخرجات أنَّ التحديث قد تمّ

لإضافة عناصر إلى القواميس أو تعديل القيم، يمكن إمّا استخدام الصياغة `dict[key] = value`، أو التابع `dict.update()`.

3. حذف عناصر من القاموس

كما يمكنك إضافة أزواج قيمة-مفتاح إلى القاموس، أو تغيير قيمه، يمكنك أيضًا حذف العناصر الموجودة في القاموس.

لتزيل زوج قيمة-مفتاح من القاموس، استخدم الصياغة التالية:

```
del dict[key]
```

لنأخذ القاموس jesse الذي يمثل أحد المستخدمين، ولنفترض أنَّ jesse لم تعد تستخدم المنصة لأجل ممارسة الألعاب، لذلك سنزيل العنصر المرتبط بالمفتاح points. بعد ذلك، سنطبع القاموس لتأكيد حذف العنصر:

```
jesse = {'username': 'JOctopus', 'online': False, 'points':
723, 'followers': 481}

del jesse['points']

print(jesse)
#  {'online': False, 'username': 'JOctopus', 'followers': 481}
```

يزيل السطر `del jesse ['points']` الزوج `'points': 723` من القاموس jesse.

إذا أردت محو جميع عناصر القاموس، فيمكنك ذلك باستخدام التابع `dict.clear()`. سيَبقى هذا القاموس في الذاكرة، وهذا مفيد في حال احتجنا إلى استخدامه لاحقًا في البرنامج، بيْد أنه سيُفرِّغ جميع العناصر من القاموس.

دعنا نزيل كل عناصر القاموس jesse:

```
jesse = {'username': 'JOctopus', 'online': False, 'points':
723, 'followers': 481}

jesse.clear()

print(jesse)    #  {}
```

تُظهِر المخرجات أنَّ القاموس صار فارغًا الآن.

إذا لم تعد بحاجة إلى القاموس، فاستخدم del للتخلص منه بالكامل:

```
del jesse

print(jesse)
```

إذا نفّذت الأمر `print()` بعد حذف القاموس `jesse`، سوف تتلقى الخطأ التالي:

```
NameError: name 'jesse' is not defined
```

4. خلاصة الفصل

ألقينا في هذا الفصل نظرة على النوع `dictionary` (القواميس) في بايثون. تتألف القواميس (dictionaries) من أزواج قيمة-مفتاح، وتوفر حلًّا ممتازًا لتخزين البيانات دون الحاجة إلى فهرستها. يتيح لنا ذلك استرداد القيم بناءً على معانيها وعلاقتها بأنواع البيانات الأخرى.

إن لم تطلع على فصل فهم أنواع البيانات، فيمكنك الرجوع إليه للتعرف على أنواع البيانات الأخرى الموجودة في بايثون.

14

التعليمات الشرطية

لا تخلو لغة برمجة من التعليمات الشرطية (conditional statement) التي تُنفَّذ بناءً على تحقق شرط معين، وهي تعليمات برمجية يمكنها التحكم في تنفيذ شيفرات معينة بحسب تحقق شرط ما من عدمه في وقت التنفيذ.

تُنفَّذ تعليمات برامج بايثون من الأعلى إلى الأسفل، مع تنفيذ كل سطر بحسب ترتيبه. باستخدام التعليمات الشرطية، يمكن للبرامج التحقق من استيفاء شروط معينة، ومن ثم تنفيذ الشيفرة المقابلة.

هذه بعض الأمثلة التي سنستخدم فيها التعليمات الشرطية:

- إن حصلت طالبة على أكثر من 65٪ في الامتحان، فأعلن عن نجاحها؛ وإلا، فأعلن عن رسوبها
- إذا كان لديه مال في حسابه، فاحسب الفائدة. وإلا، فاحسب غرامة
- إن اشتروا 10 برتقالات أو أكثر، فاحسب خصمًا بمقدار 5٪؛ وإلا فلا تفعل

تقيِّم الشيفرة الشرطية شروطًا، ثم تُنفِّذ شيفرةً بناءً على ما إذا تحققت تلك الشروط أم لا. ستتعلم في هذا الفصل كيفيَّة كتابة التعليمات الشرطية في بايثون.

1. التعليمة if

سنبدأ بالتعليمة `if`، والتي تتحقق مما إذا تحقق شرط محدَّد أم لا، وفي حال تحقق الشرط، فستنفَّذ الشيفرة المقابلة له. لنبدأ بأمثلة عملية توضح ذلك. افتح ملفًا، واكتب الشيفرة التالية:

```
grade = 70
if grade >= 65:
   print("درجة النجاح")
```

أعطينا للمتغير grade القيمة 70. ثمَّ استخدمنا التعليمة if لتقييم ما إذا كان المتغير grade أكبر من (=<) أو يساوي 65. وفي تلك الحالة، سيطبع البرنامج السلسلة النصية التالية: "درجة النجاح".

احفظ البرنامج بالاسم grade.py، ثم نفِّذه في بيئة البرمجة المحلية من نافذة الطرفية باستخدام الأمر python grade.py. في هذه الحالة، الدرجة 70 تلبي الشرط، لأنّها أكبر من 65، لذلك ستحصل على المخرجات التالية عند تنفيذ البرنامج:

```
درجة النجاح
```

لنغيّر الآن نتيجة هذا البرنامج عبر تغيير قيمة المتغير grade إلى 60:

```
grade = 60

if grade >= 65:
   print("درجة النجاح")
```

بعد حفظ وتنفيذ الشيفرة، لن نحصل على أي مخرجات، لأنّ الشرط لم يتحقق، ولم نأمر البرنامج بتنفيذ تعليمة أخرى.

مثال آخر، دعنا نتحقَّق مما إذا كان رصيد الحساب المصرفي أقل من 0. لننشئ ملفا باسم account.py، ونكتب البرنامج التالي:

```
balance = -5

if balance < 0:
   print("الحساب فارغ، أضف مبلغا الآن، أو ستحصل على غرامة.")
```

عند تنفيذ البرنامج باستخدام python account.py، سنحصل على المخرجات التالية:

```
الحساب فارغ، أضف مبلغًا الآن، أو ستحصل على غرامة.
```

أعطينا للمتغير balance القيمة ‎-5، وهي أقل من 0 في البرنامج السابق. ولمّا كان الرصيد مستوفيًا لشرط التعليمة if (أي balance < 0)، فسنحصل على سلسلة نصية في المخرجات بمجرد حفظ الشيفرة وتنفيذها. مرة أخرى، لو غيرنا الرصيد إلى القيمة 0 أو إلى عدد موجب، فلن نحصل على أيّ مخرجات.

2. التعليمة else

قد تريد من البرنامج أن يفعل شيئًا ما في حال عدم تحقق شرط التعليمة if. في المثال أعلاه، نريد طباعة مخرجات في حال النجاح والرسوب. ولفعل ذلك، سنضيف التعليمة else إلى شرط الدرجة أعلاه وفق الصياغة التالية:

```
grade = 60

if grade >= 65:
    print("درجة النجاح")

else:
    print("درجة الرسوب")
```

قيمة المتغير grade تساوي 60، لذلك فشرط التعليمة if غير متحقق، وبالتالي فإنَّ البرنامج لن يطبع السلسلة "درجة النجاح". تخبر التعليمة else البرنامجَ أنَّ عليه طباعة السلسلة النصية "درجة الرسوب". عندما نحفظ البرنامج وننفِّذه، سنحصل على المخرجات التالية:

```
درجة الرسوب
```

إذا عدّلنا البرنامج وأعطينا المتغيرَ grade القيمة 65 أو أعلى منها، فسنحصل بدلًا من ذلك على الناتج "درجة النجاح".

لإضافة التعليمة else إلى مثال الحساب المصرفي، سنعيد كتابة الشيفرة كما يلي:

```
balance = 522

if balance < 0:
   print("الحساب فارغ، أضف مبلغا الآن، أو ستحصل على غرامة")

else:
   print("رصيدك أكبر من 0.")
```

سنحصل على المخرجات التالية:

```
رصيدك أكبر من 0.
```

هنا، غيّرنا قيمة المتغير balance إلى عدد موجب لكي تُنفَّذ الشيفرة المقابلة للتعليمة else. إن أردت تنفيذ الشيفرة المقابلة للتعليمة if، غيِّر القيمة إلى عدد سالب.

من خلال دمج العبارتين if و else، فأنت تنشئ تعليمة شرطية مزدوجة، والتي ستجعل الحاسوب ينفذ شيفرة برمجية معينة سواء تم استيفاء شرط if أم لا.

3. التعليمة else if

حتى الآن، عملنا على تعليمات شرطية ثنائية، أي إن تحقق الشرط، فنفذ شيفرة ما، وإلا، فنفِّذ شيفرة أخرى فقط. لكن في بعض الحالات، قد تريد برنامجًا يتحقق من عدة حالات شرطية. ولأجل هذا، سوف نستخدم التعليمة else if، والتي تُكتب في بايثون هكذا elif. تشبه التعليمة elif - أو else if - التعليمة if، ومهمتها التحقق من شرط إضافي آخر.

في برنامج الحساب المصرفي، قد نرغب في الحصول على ثلاثة مخرجات مختلفة مقابلة لثلاث حالات مختلفة:

- الرصيد أقل من 0
- الرصيد يساوي 0
- الرصيد أعلى من 0

ستوضع التعليمة elif بين التعليمة if والتعليمة else كما يلي:

```
. . .
if balance < 0:
    print("الحساب فارغ، أضف مبلغا الآن، أو ستحصل على غرامة")

elif balance == 0:
    print("الرصيد يساوي 0، أضف مبلغًا قريبًا.")

else:
    print("رصيدك أكبر من 0.")
```

الآن، هناك ثلاثة مخرجات محتملة يمكن أن تُطبع عند تنفيذ البرنامج:

- إن كان المتغير balance يساوي 0، فسنحصل على المخرجات من التعليمة elif (أي السلسلة "الرصيد يساوي 0، أضف مبلغًا قريبًا.").

- إذا ضُبِط المتغير balance عند عدد موجب، فسوف نحصل على المخرجات من التعليمة else (أي طباعة السلسلة "رصيدك أكبر من 0.").

- إذا ضُبِط المتغير balance عند عدد سالب، فسنحصل على المخرجات من التعليمة if (أي السلسلة "الحساب فارغ، أضف مبلغا الآن، أو ستحصل على غرامة").

ماذا لو أردنا أن نأخذ بالحسبان أكثر من ثلاثة احتمالات؟ يمكننا كتابة عدة تعليمات `elif` في الشيفرة البرمجية.

لنُعِد كتابة البرنامج `grade.py` بحيث يقابل كل نطاق من الدرجات علامة محددة:

- 90 أو أعلى تكافئ الدرجة A
- 80-89 تعادل الدرجة +B
- 70-79 تعادل الدرجة B
- 65-69 تعادل الدرجة -B
- 64 أو أقل تكافئ الدرجة F

سنحتاج لتنفيذ هذه الشيفرة إلى تعليمة `if` واحد، وثلاث تعليمات `elif`، وتعليمة `else` تعالج جميع الحالات الأخرى.

دعنا نعيد كتابة الشيفرة من المثال أعلاه لطباعة سلسلة نصية مقابلة لكل علامة. يمكننا الإبقاء على التعليمة `else` كما هي.

```
. . .
if grade >= 90:
    print("A")

elif grade >=80:
    print("B+")

elif grade >=70:
    print("B")
```

```
elif grade >= 65:
    print("B-")

else:
    print("F")
```

تُنفّذ التعليمات `elif` بالترتيب. هذا البرنامج سيكمل الخطوات التالية:

- إذا كانت الدرجة أكبر من 90، فسيطبع البرنامجُ A، وإذا كانت الدرجة أقل من 90، فسيمرّ البرنامج إلى التعليمة التالية …
- إذا كانت الدرجة أكبر من أو تساوي 80، فسيطبع البرنامجُ +B، إذا كانت الدرجة تساوي 79 أو أقل، فسيمرّ البرنامج إلى التعليمة التالية …
- إذا كانت الدرجة أكبر من أو تساوي 70، فسيطبعُ البرنامجُ B، إذا كانت الدرجة تساوي 69 أو أقل، فسيمرّ البرنامج إلى التعليمة التالية …
- إذا كانت الدرجة أكبر من أو تساوي 65، فسيطبعُ البرنامجُ -B، وإذا كانت الدرجة تساوي 64 أو أقل، فسيمرّ البرنامج إلى التعليمة التالية …
- سيطبع البرنامج F، لأنه لم يتم استيفاء أيٍّ من الشروط المذكورة أعلاه.

4. تعليمات if المتشعبة

بعد أن تتعود على التعليمات `if` و `elif` و `else`، يمكنك الانتقال إلى التعليمات الشرطية المتشعبة (nested conditional statements).

يمكننا استخدام تعليمات `if` المتشعِّبة في الحالات التي نريد فيها التحقق من شرط ثانوي بعد التأكُّد من تحقق الشرط الرئيسي. لهذا، يمكننا حشر تعليمة `if-else` داخل تعليمة `if-else` أخرى. لنلقِ نظرة على صياغة `if` المتشعبة:

```
if statement1:                  # تعليمة if الخارجية
    print("true")

    if nested_statement:        # تعليمة if المتشعبة
        print("yes")

    else:                       # تعليمة else المتشعبة
        print("no")

else:                           # تعليمة else الخارجية
    print("false")
```

هناك عدة مخرجات محتملة لهذه الشيفرة:

- إذا كانت statement1 صحيحة، فسيتحقق البرنامج مما إذا كانت nested_statement صحيحة أيضًا. إذا كانت كلتا الحالتين صحيحتان، فسنحصل على المخرجات التالية:

```
true
yes
```

- ولكن إن كانت statement1 صحيحة، و nested_statement خطأ، فسنحصل على المخرجات التالية:

```
true
no
```

- وإذا كانت statement1 خطأ، فلن تُنفّذ تعليمة if-else المتشعّبة على أيِّ حال، لذلك ستُنفّذ التعليمة else وحدها، والمخرجات ستكون:

```
false
```

يمكن أيضًا استخدام عدة تعليمات if متشعبة في الشيفرة:

```
if statement1:              # الخارجية if
    print("hello world")

    if nested_statement1:   # المتشعبة الأولى if
        print("yes")

    elif nested_statement2: # المتشعبة الأولى elif
        print("maybe")

    else:                   # المتشعبة الأولى else
        print("no")

elif statement2:            # الخارجية elif
    print("hello galaxy")

    if nested_statement3:   # المتشعبة الثانية if
        print("yes")

    elif nested_statement4: # المتشعبة الثانية elif
        print("maybe")

    else:                   # المتشعبة الثانية else
        print("no")

else:                       # الخارجية else
    statement("Hello")
```

في الشيفرة البرمجية أعلاه، هناك تعليمات if و elif متشعِّبة داخل كل تعليمات if. هذا سيفسح المجال لمزيد من الخيارات في كل حالة.

دعنا نلقي نظرة على مثال لتعليمات if متشعبة في البرنامج grade.py. يمكننا التحقق أولًا مما إذا حقَّق الطالب درجة النجاح (أكبر من أو تساوي 65٪)، ثم نحدِّد العلامة المقابلة للدرجة. إذا لم يحقِّق الطالب درجة النجاح، فلا داعي للبحث عن العلامة المقابلة للدرجة، وبدلًا من ذلك، يمكن أن نجعل البرنامج يطبع سلسلة نصية فيها إعلان عن رسوب الطالب.

ستبدو الشيفرة المعدلة عن المثال السابق كما يلي:

```
. . .
if grade >= 65:
    print("درجة النجاح:")

    if grade >= 90:
        print("A")

    elif grade >=80:
        print("B+")

    elif grade >=70:
        print("B")

    elif grade >= 65:
        print("B-")

else:
    print("F")
```

إذا أعطينا للمتغير grade القيمة 92، فسيُستوفى الشرط الأول، وسيَطبع البرنامجُ "درجة النجاح:". بعد ذلك، سيتحقَّق مما إذا كانت الدرجة أكبر من أو تساوي 90، وبما أنَّ هذا الشرط متحقق أيضًا، فستُطبع A.

أمَّا إذا أعطينـا للمتغيـر grade القيمـة 60، فلن يُسـتوفَى الشـرط الأول، لـذلك سـيتخطى البرنامج تعليمات if المتشعّبة، وينتقل إلى التعليمة else، ويطبع F.

يمكننا بالطبع إضافة المزيد من الخيارات، واستخدام طبقة ثانية من تعليمات if المتشعّبة. ربما نود إضافة الدرجات التفصيلية +A و A و -A. يمكننا القيام بذلك عن طريق التحقق أولًا من اجتيـاز درجة النجـاح، ثم التحقُّق مما إذا كـانت الدرجة تسـاوي 90 أو أعلى، ثم التحقق مما إذا كانت الدرجة تتجاوز 96، وفي تلك الحالة ستقابل العلامة +A. اطلع على المثال التالي:

```
. . .
if grade >= 65:
    print("درجة النجاح:")

    if grade >= 90:
        if grade > 96:
            print("A+")

elif grade > 93 and grade <= 96:

            print("A")

        elif grade >= 90:
            print("A-")
. . .
```

في الشيفرة أعلاه، في حال تعيين المتغير grade عند القيمة 96، سيقوم البرنامج بما يلي:

- التحقق مما إذا كانت الدرجة أكبر من أو تساوي 65 (صحيح)
- طباعة السلسلة "درجة النجاح: "
- التحقق مما إذا كانت الدرجة أكبر من أو تساوي 90 (صحيح)

- التحقق مما إذا كانت الدرجة أكبر من 96 (خطأ)
- التحقق مما إذا كانت الدرجة أكبر من 93، وأقل من أو تساوي 96 (صحيح)
- طباعة A
- تجاوز التعليمات الشرطية المتشعبة وتنفيذ باقي الشيفرة

ستكون مخرجات البرنامج في حال كانت الدرجة تساوي 96 كالتالي:

```
درجة النجاح:
A
```

تساعد تعليمات `if` المتشعبة على إضافة عدة مستويات من الشروط الفرعية إلى الشيفرة.

5. خلاصة الفصل

ستتحكم باستخدام التعليمات الشرطية، مثل التعليمة `if`، في مسار البرنامج أي تدفق تنفيذ الشيفرة. تطلب التعليمات الشرطية من البرنامج التحقُّق من استيفاء شرط معيَّن من عدمه. وإذا تم استيفاء الشرط، فستُنفَّذ شيفرة معينة، وإلا فسيستمر تنفيذ البرنامج وينتقل إلى الأسطر التالية.

يمكنك الدمج بين التعليمات الشرطية والمعاملات المنطقية، بما فيها `and` و `or`، واستخدام التعليمات الشرطية مع الحلقات التكرارية.

15

المهام التكرارية: مدخل إلى الحلقات

نستفيد من البرامج الحاسوبية خير استفادة في أتمتة المهام وإجراء المهام التكرارية لكيلا نحتاج إلى القيام بها يدويًا، وإحدى طرائق تكرار المهام المتشابهة هي استخدام حلقات التكرار (loops)، وسنشرح في هذا الفصل حلقتي التكرار الشهيرتين في بايثون -وسائر لغات البرمجة- while و for، وكيفية استعمالها.

1. حلقة التكرار while

حلقة التكرار while تؤدي إلى تكرار تنفيذ قسم من الشيفرة بناءً على متغير منطقي (boolean)، وسيستمر تنفيذ هذه الشيفرة لطالما كانت نتيجة التعبير المستعمل معها تساوي true أي طالما كان شرطٌ ما محققًّا.

يمكنك أن تتخيل أنَّ حلقة while هي عبارة شريطة تكرارية؛ فبعد انتهاء تنفيذ التعليمة الشرطية if، يُستَكمَل تنفيذ بقية البرنامج، لكن مع حلقة while فسيعود تنفيذ البرنامج إلى بداية الحلقة بعد انتهاء تنفيذها إلى أن يصبح الشرط مساويًا للقيمة false أي لم يعد الشرط محقَّقًا.

وعلى النقيض من حلقات for التي تُنفَّذ عددًا معيَّنًا من المرات، فسيستمر تنفيذ حلقات while اعتمادًا على شرطٍ معيَّن، لذا لن تحتاج إلى عدد مرات تنفيذ الحلقة قبل إنشائها.

الشكل العام لحلقات while في لغة بايثون كالآتي:

```
while [a condition is True]:
    [do something]
```

سيستمر تنفيذ التعليمات البرمجية الموجودة داخل الحلقة إلى أن يُقيَّم الشرط الذي يلي while إلى القيمة false.

لنُنشِئ برنامجًا صغيرًا فيه حلقة while، ففي هذه البرنامج سنطلب من المستخدم إدخال كلمة مرور. وهنالك خياران أمام حلقة التكرار:

- إمّا أن تكون كلمة المرور صحيحة، فعندها سينتهي تنفيذ حلقة while.
- أو أن تكون كلمة المرور غير صحيحة، فعندها سيستمر تنفيذ حلقة التكرار.

لنُنشِئ ملفًا باسم password.py في محرِّرنا النصي المفضَّل، ولنبدأ بتهيئة المتغير paasword بإسناد سلسلة نصية فارغة إليه:

```
password = ''
```

نستخدم المتغير السابق للحصول على مدخلات المستخدم داخل حلقة التكرار while.

علينا بعد ذلك إنشاء حلقة while مع تحديد ما هو الشرط الذي يجب تحقيقه:

```
password = ''

while password != 'password':
```

أتبَعنا -في المثال السابق- الكلمة المحجوزة while بالمتغير password، ثمَّ سنتحقق إذا كانت قيمة المتغير password تساوي السلسلة النصية 'password' (لا تنسَ أنَّ قيمة المتغير سنحصل عليها من مدخلات المستخدم)، يمكنك أن تختار أي سلسلة نصية تشاء لموازنة مدخلات المستخدم بها. هذا يعني أنَّه لو أدخل المستخدم السلسلة النصية password فستتوقف حلقة التكرار وسيُكمَل تنفيذ البرنامج وستُنفَّذ أيّة شيفرات خارج الحلقة، لكن إذا أدخل المستخدم أيّة سلسلة نصية لا تساوي password فسيُكمَل تنفيذ الحلقة. علينا بعد ذلك إضافة الشيفرة المسؤولة عمّا يحدث داخل حلقة while:

```
password = ''

while password != 'password':
    print('What is the password?')
    password = input()
```

نفَّذ البرنامج عبارة print داخل حلقة while والتي تسأل المستخدم عن كلمة مروره، ثم أسندنا قيمة مدخلات المستخدم (التي حصلنا عليها عبر الدالة ()input) إلى المتغير password. سيتحقَّق البرنامج إذا كانت قيمة المتغير password تساوي السلسلة النصية 'password'، وإذا تحقَّق ذلك فسينتهي تنفيذ حلقة while. لنضف سطرًا آخر إلى البرنامج لنعرف ماذا يحدث إن أصبحت قيمة الشرط مساويةً إلى false:

```
password = ''

while password != 'password':
    print('What is the password?')
    password = input()

print('Yes, the password is ' + password + '. You may enter.')
```

لاحظ أنَّ آخر عبارة ()print موجودة خارج حلقة while، لذا عندما يُدخِل المستخدم الكلمة password عند سؤاله عن كلمة مروره، فستُطبَع آخر جملة والتي تقع خارج حلقة التكرار. لكن ماذا يحدث لو لم يدخل المستخدم الكلمة password قط؟ إذ لن يستمر تنفيذ البرنامج ولن يروا آخر عبارة ()print وسيستمر تنفيذ حلقة التكرار إلى ما لا نهاية! يستمر تنفيذ حلقة التكرار إلى ما لا نهاية إذا بقي تنفيذ البرنامج داخل حلقة تكرار دون الخروج منها. وإذا أردت الخروج من حلقة تكرار نهائية، فاضغط Ctrl+C في سطر الأوامر. احفظ البرنامج ثم شغِّله:

```
python password.py
```

سيُطلَب منك إدخال كلمة المرور، ويمكنك تجربة ما تشاء من الكلمات. هذا مثالٌ عن ناتج البرنامج:

```
What is the password?
hello
What is the password?
sammy
What is the password?
PASSWORD
What is the password?
password
Yes, the password is password. You may enter.
```

أبقِ في ذهنك أنَّ السلاسل النصية حساسة لحالة الأحرف إلا إذا استعملتَ دالةً من دوال النصوص لتحويل السلسلة النصية إلى حالة الأحرف الصغيرة (على سبيل المثال) قبل التحقُّق منها.

ا. تطبيق عملي

بعد أن تعلمنا المبدأ الأساسي لحلقة تكرار `while`، فلنُنشِئ لعبة تعمل على سطر الأوامر لتخمين الأرقام والتي تستعمل الحلقة `while` . نريد من الحاسوب أن يُنشِئ أرقامًا عشوائيةً لكي يحاول المستخدمون تخمينها، لذا علينا استيراد الوحدة `random` عبر استخدام التعليمة `import` (سنطرق لاحقًا إلى كيفية استيراد الوحدات في فصل الوحدات)، وإذا لم تكن هذه الحزمة مألوفةً لك فيمكنك قراءة المزيد من المعلومات عن توليد الأرقام العشوائية في توثيق بايثون.

لنُنشِئ بدايةً ملفًا باسم `guess.py` في محرر النصي المفضَّل:

```
import random
```

علينا الآن إسناد عدد صحيح عشوائي إلى المتغير number، ولنجعل مجاله من 1 إلى 25 (بما فيها تلك الأرقام) كيلا نجعل اللعبة صعبة جدًا.

```
import random

number = random.randint(1, 25)
```

يمكننا الآن إنشاء حلقة while، وذلك بتهيئة متغير ثم كتابة الحلقة:

```
import random

number = random.randint(1, 25)

number_of_guesses = 0

while number_of_guesses < 5:
    print('Guess a number between 1 and 25:')

    guess = input()
    guess = int(guess)

    number_of_guesses = number_of_guesses + 1

    if guess == number:
      break
```

هيأنا متغيرًا اسمه number_of_guesses قيمته 0، وسوف نزيد قيمته عند كل تكرار للحلقة لكي لا تصبح حلقتنا لا نهائية ثم سنضيف حلقة while التي تشترط ألّا تزيد قيمة المتغير number_of_guesses عن خمس تكرارات.

وبعد المحاولة الخامسة سيُعاد المستخدم إلى سطر الأوامر، وإذا حاول المستخدم إدخال أيّ شيء غير رقمي فسيحصل على رسالة خطأ.

أضفنا داخل حلقة `while` عبارة `print()` لطلب إدخال رقم من المستخدم، ثم سنأخذ مدخلات المستخدم عبر الدالة `input()` ونُسنِدَها إلى المتغير `guess`، ثم سنحوِّل المتغير `guess` من سلسلة نصية إلى عدد صحيح. وقبل انتهاء حلقة التكرار، فعلينا زيادة قيمة المتغير `number_of_guesses` بمقدار 1، لكيلا تُنفَّذ حلقة التكرار أكثر من 5 مرات.

وفي النهاية، كتبنا التعليمة `if` شرطية لنرى إذا كان المتغير `guess` الذي أدخله المستخدم مساوٍ للرقم الموجود في المتغير `number` الذي ولَّده الحاسوب، وإذا تحقق الشرط فسنستخدم التعليمة `break` للخروج من الحلقة. أصبح البرنامج جاهزًا للاستخدام، ويمكننا تشغيله عبر تنفيذ الأمر التالي:

```
python guess.py
```

صحيحٌ أنَّ البرنامج يعمل عملًا سليمًا، لكن المستخدم لن يعلم إذا كان تخمينه صحيحًا ويمكنه أن يخمِّن الرقم خمس مرات دون أن يعلم إذا كانت إحدى محاولاته صحيحة. هذا مثال عن مخرجات البرنامج:

```
Guess a number between 1 and 25:
11
Guess a number between 1 and 25:
19
Guess a number between 1 and 25:
22
Guess a number between 1 and 25:
3
```

```
Guess a number between 1 and 25:
8
```

لنضف بعض التعليمات الشرطية خارج حلقة التكرار لكي يحصل المستخدم على معلومات فيما إذا استطاعوا تخمين الرقم أم لا، وسنضيف هذه العبارات في نهاية الملف:

```
import random

number = random.randint(1, 25)

number_of_guesses = 0

while number_of_guesses < 5:
   print('Guess a number between 1 and 25:')
   guess = input()
   guess = int(guess)

   number_of_guesses = number_of_guesses + 1

   if guess == number:
     break

if guess == number:
   print('You guessed the number in ' + str(number_of_guesses)
+ ' tries!')

else:
   print('You did not guess the number. The number was ' +
str(number))
```

سيُخبِر البرنامجُ في هذه المرحلة المستخدمَ إذا استطاعوا تخمين الرقم، لكن ذلك لن يحدث إلا بعد انتهاء حلقة التكرار وبعد انتهاء عدد مرات التخمين المسموحة. ولمساعد

المستخدم قليلًا، فلنضف بعض التعليمات الشرطية داخل حلقة `while` وستخبر تلك التعليمات المستخدم إذا كان تخمينه أعلى من الرقم أو أصغر منه، لكي يستطيعوا تخمين الرقم بنجاح، وسنضيف تلك التعليمات الشرطية قبل السطر الذي يحتوي على `if guess == number`:

```
import random

number = random.randint(1, 25)
number_of_guesses = 0

while number_of_guesses < 5:
   print('Guess a number between 1 and 25:')
   guess = input()
   guess = int(guess)

   number_of_guesses = number_of_guesses + 1

   if guess < number:
     print('Your guess is too low')

   if guess > number:
     print('Your guess is too high')

   if guess == number:
     break

if guess == number:
   print('You guessed the number in ' + str(number_of_guesses)
+ ' tries!')

else:
   print('You did not guess the number. The number was ' +
str(number))
```

وعندما نُشغِّل البرنامج مرةً أخرى بتنفيذ guess.py python، فيمكننا ملاحظة أنَّ المستخدم سيحصل على بعض المساعدة، فلو كان الرقم المولَّد عشوائيًا هو 12 وكان تخمين المستخدم 18، فسيُخبره البرنامج أنَّ الرقم الذي خمنه أكبر من الرقم العشوائي، وذلك لكي يستطيع تعديل تخمينه وفقًا لذلك. هنالك الكثير من التحسينات التي يمكن إجراؤها على الشيفرة السابقة، مثل تضمين آلية لمعالجة الأخطاء التي تحدث عندما لا يُدخِل المستخدم عددًا صحيحًا، لكن كان غرضنا هو رؤية كيفية استخدام حلقة while في برنامج قصير ومفيد يعمل من سطر الأوامر.

2. حلقة التكرار for

حلقة for تؤدي إلى تكرار تنفيذ جزء من الشيفرات بناءً على عدَّاد أو على متغيِّر، وهذا يعني أنَّ حلقات for تستعمل عندما يكون عدد مرات تنفيذ حلقة التكرار معلومًا قبل الدخول في الحلقة، وذلك على النقيض من حلقات while المبنية على شرط.

تُبنى حلقات for في بايثون كما يلي:

```
for [iterating variable] in [sequence]:
   [do something]
```

ستُنفَّذ الشيفرات الموجودة داخل حلقة التكرار عدّة مرات إلى أن تنتهي الحلقة. لننظر إلى كيفية مرور الحلقة for على مجالٍ من القيم:

```
for i in range(0,5):
  print(i)
```

سيُخرِج البرنامج السابق عند تشغيله الناتج الآتي:

```
0
1
2
3
4
```

ضبطنا المتغير `i` في حلقة `for` ليحتوي على القيمة التي ستُنفَّذ عليها حلقة التكرار، وكان مجال القيم التي ستُسنَد إلى هذا المتغير من 0 إلى 5. ثم طبعًا قيمة المتغير في كل دوران لحلقة التكرار، لكن أبقِ في ذهنك أنَّنا نميل إلى بدء العد من الرقم 0 في البرمجة، وعلى الرغم من عرض خمسة أرقام، لكنها تبدأ بالرقم 0 وتنتهي بالرقم 4. من الشائع أن ترى استخدامًا لحلقة `for` عندما تحتاج إلى تكرار كتلة معيَّنة من الشيفرات لعددٍ من المرات.

ا. استخدام حلقة التكرار for مع الدالة range()

إحدى أنواع السلاسل غير القابلة للتعديل في بايثون هي تلك الناتجة من الدالة `range()`، وتستخدم الدالة `range()` في حلقات التكرار للتحكم بعدد مرات تكرار الحلقة. عند التعامل مع الدالة `range()` عليك أن تمرر معاملًا رقميًا أو معاملين أو ثلاثة معاملات:

- `start`: يشير إلى القيم العددية الصيحية التي ستبدأ بها السلسلة، وإذا لم تُمرَّر قيمة لهذا المعامل فستبدأ السلسلة من 0.
- `stop`: هذا المعامل مطلوب دومًا وهو القيمة العددية الصحيحة التي تمثل نهاية السلسلة العددية لكن دون تضمينها.
- `step`: هي مقدار الخطوة، أي عدد الأرقام التي يجب زيادتها (أو إنقاصها إن كنَّا نتعامل مع أرقام سالبة) في الدورة القادمة، وقيمة المعامل `step` تساوي 1 إن لم تُحدَّد له قيمة.

لننظر إلى بعض الأمثلة التي نُمرِّر فيها مختلف المعاملات إلى الدالة `range()`. لنبدأ بتمرير المعامل `stop` فقط، أي أنَّ السلسلة الآتية من الشكل `range(stop)`:

```
for i in range(6):
  print(i)
```

كانت قيمة المعامل `stop` في المثال السابق مساويةً للرقم 6، لذا ستمر حلقة التكرار من بداية المجال 0 إلى نهايته 6 (باستثناء الرقم 6 كما ذكرنا أعلاه):

```
0
1
2
3
4
5
```

المثال الآتي من الشكل `range(start ,stop)` الذي تُمرَّر قيم بدء السلسلة ونهايتها:

```
for i in range(20,25):
  print(i)
```

المجال -في المثال السابق- يتراوح بين 20 (بما فيها الرقم 20) إلى 25 (باستثناء الرقم 25)، لذا سيبدو الناتج كما يلي:

```
20
21
22
23
24
```

الوسيط `step` الخاص بالدالة `range()` شبيه بمعامل الخطوة الذي نستعمله عند تقسيم السلاسل النصية لأنه يستعمل لتجاوز بعض القيم ضمن السلسلة. يأتي المعامل `step` في آخر

قائمة المعاملات التي تقبلها الدالة `range()` وذلك بالشكل `range(start, stop, step)`.

لنستعمل المعامل `step` مع قيمة موجبة:

```
for i in range(0,15,3):
  print(i)
```

سيؤدي المثال السابق إلى إنشاء سلسلة من الأرقام التي تبدأ من 0 وتنتهي عند 15 لكن قيمة المعامل `step` هي 3، لذا سيتم تخطي رقمين في كل دورة، أي سيكون الناتج كالآتي:

```
0
3
6
9
12
```

يمكننا أيضًا استخدام قيمة سالبة للمعامل `step` للدوران إلى الخلف، لكن علينا تعديل قيم `start` و `stop` بما يتوافق مع ذلك:

```
for i in range(100,0,-10):
  print(i)
```

قيمة المعامل `start` في المثال السابق هي 100، وكانت قيمة المعامل `stop` هي 0، والخطوة هي `-10`، لذا ستبدأ السلسلة من الرقم 100 وستنتهي عند الرقم 0، وسيكون التناقص بمقدار 10 في كل دورة، ويمكننا ملاحظة ذلك في الناتج الآتي:

```
100
90
80
70
60
```

```
50
40
30
20
10
```

عندما نبرمج باستخدام لغة بايثون، فسنجد أننا نستفيد كثيرًا من السلاسل الرقمية التي تنتجها الدالة `range()`.

ب. استخدام حلقة for مع أنواع البيانات المتسلسلة

يمكن الاستفادة من القوائم (من النوع `list`) وغيرها من أنواع البيانات المتسلسلة واستعمالها بعدِّها معاملات لحلقات `for`، فبدلًا من الدوران باستخدام الدالة `range()` فيمكننا تعريف قائمة ثم الدوران على عناصرها. سنُسنِد في المثال الآتي قائمةً إلى متغيِّر، ثم سنستخدم حلقة `for` للدوران على عناصر القائمة:

```
sharks = ['hammerhead', 'great white', 'dogfish', 'frilled',
'bullhead', 'requiem']

for shark in sharks:
  print(shark)
```

في هذه الحالة، طبعنا كل عنصر موجود في القائمة؛ وصحيحٌ أننا استعملنا الكلمة `shark` اسمًا للمتغير، لكن يمكنك استعمال أي اسم صحيح آخر ترغب به، وستحصل على نفس النتيجة.

ناتج تنفيذ المثال السابق هو:

```
hammerhead
great white
```

```
dogfish
frilled
bullhead
requiem
```

الناتج السابق يُظهِر دوران الحلقة for على جميع عناصر القائمة مع طباعة كل عنصر في سطرٍ منفصل. يشيع استخدام القوائم والأنواع الأخرى من البيانات المتسلسلة مثل السلاسل النصية وبنى tuple (الصفوف) مع حلقات التكرار لسهولة الدوران على عناصرها. يمكنك دمج هذه الأنواع من البيانات مع الدالة range() لإضافة عناصر إلى قائمة، مثل:

```
sharks = ['hammerhead', 'great white', 'dogfish', 'frilled',
'bullhead', 'requiem']

for item in range(len(sharks)):
  sharks.append('shark')

print(sharks)
```

الناتج:

```
['hammerhead', 'great white', 'dogfish', 'frilled', 'bullhead',
'requiem', 'shark', 'shark', 'shark', 'shark', 'shark',
'shark']
```

أضفنا هنا السلسلة النصية 'shark' خمس مرات (وهو نفس طول القائمة sharks الأصلي) إلى القائمة sharks.

يمكننا استخدام حلقة for لبناء قائمة جديدة:

```
integers = []
```

```
for i in range(10):
  integers.append(i)
print(integers)
```

هيّئنا في المثال السابق قائمةً فارغةً باسم integers لكن حلقة التكرار for ملأت القائمة لتصبح كما يلي:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

وبشكلٍ شبيهٍ بما سبق، يمكننا الدوران على السلاسل النصية:

```
sammy = 'Sammy'

for letter in sammy:
   print(letter)
```

الناتج:

```
S
a
m
m
y
```

يمكن الدوران على بنى tuple كما هو الحال في القوائم والسلاسل النصية. عند المرور على عناصر نوع البيانات dictionary، فمن المهم أن تبقي بذهنك البنية الخاصة به «مفتاح:قيمة» (key:value) لكي تضمن أنَّك تستدعي العنصر الصحيح من المتغير.

إليك مثالٌ بسيطٌ نعرض فيه المفتاح (key) والقيمة (value):

```
sammy_shark = {'name': 'Sammy', 'animal': 'shark', 'color':
'blue', 'location': 'ocean'}
```

```
for key in sammy_shark:
   print(key + ': ' + sammy_shark[key])
```

الناتج:

```
name: Sammy
animal: shark
location: ocean
color: blue
```

عند استخدام متغيرات من النوع dictionary (قاموس) مع حلقات for فيكون المتغير المرتبط بحلقة التكرار متعلقًا بمفتاح القيم، وعلينا استخدام الصياغة dictionary_variable[iterating_variable] للوصول إلى القيمة الموافقة للمفتاح. ففي المثال السابق كان المتغير المرتبط بحلقة التكرار باسم key وهو يُمثِّل المفاتيح، واستعملنا sammy_shark[key] للوصول إلى القيمة المرتبطة بذاك المفتاح. خلاصة ما سبق، تُستعمَل حلقات التكرار عادةً للدوران على عناصر البيانات المتسلسلة وتعديلها.

ج. حلقات for المتشعِّبة

يمكن تشعيب حلقات التكرار في بايثون، كما هو الحال في بقية لغات البرمجة. حلقة التكرار المتشعِّبة هي الحلقة الموجودة ضمن حلقة تكرار أخرى، وهي شبيهة بعبارات if المتشعِّبة. تُبنى حلقات التكرار المتشعبة كما يلي:

```
# الحلقة الخارجية
for [first iterating variable] in [outer loop]:
    [do something]  # اختياري
```

```
    # الحلقة الداخلية الفرعية
    for [second iterating variable] in [nested loop]:
        [do something]
```

يبدأ البرنامج بتنفيذ حلقة التكرار الخارجية، ويُنفَّذ أوّل دوران فيها، وأوَّل دوران سيؤدي إلى الدخول إلى حلقة التكرار الداخلية، مما يؤدي إلى تنفيذها إلى أن تنتهي تمامًا. ثم سيعود تنفيذ البرنامج إلى بداية حلقة التكرار الخارجية، ويبدأ بتنفيذ الدوران الثاني، ثم سيصل التنفيذ إلى حلقة التكرار الداخلية، وستُنفَّذ حلقة التكرار الداخلية بالكامل، ثم سيعود التنفيذ إلى بداية حلقة التكرار الخارجية، وهلم جرًا إلى أن ينتهي تنفيذ حلقة التكرار الخارجية أو إيقاف حلقة التكرار عبر استخدام التعليمة break أو غيرها. لنُنشِئ مثالًا يستعمل حلقة for متشعبة لكي نفهم كيف تعمل بدقة، إذ ستمر حلقة التكرار الخارجية في المثال الآتي على قائمة من الأرقام اسمها num_list، أما حلقة التكرار الداخلية فستمر على قائمة من السلاسل النصية اسمها alpha_list:

```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']

for number in num_list:
    print(number)
    for letter in alpha_list:
        print(letter)
```

سيظهر الناتج الآتي عند تشغيل البرنامج:

```
1
a
b
c
```

```
2
a
b
c
3
a
b
c
```

يُظهِر الناتج السابق أنَّ البرنامج أكمل أوّل دوران على عناصر حلقة التكرار الخارجية بطباعة الرقم 1، ومن ثم بدأ تنفيذ حلقة التكرار الدخلية مما يطبع الأحرف a و b و c على التوالي. وبعد انتهاء تنفيذ حلقة التكرار الداخلية، عاد البرنامج إلى بداية حلقة التكرار الخارجية طابعًا الرقم 2، ثم بدأ تنفيذ حلقة التكرار الداخلية (مما يؤدي إلى إظهار a و b و c مجددًا). وهكذا دواليك.

يمكن الاستفادة من حلقات for المتشعبة عند المرور على عناصر قوائم تتألف من قوائم. فلو استعملنا حلقة تكرار وحيدة لعرض عناصر قائمة تتألف من عناصر تحتوي على قوائم، فستُعرَض قيم القوائم الداخلية:

```
list_of_lists = [['hammerhead', 'great white', 'dogfish'],[0,
1, 2],[9.9, 8.8, 7.7]]

for list in list_of_lists:
    print(list)
```

الناتج:

```
['hammerhead', 'great white', 'dogfish']
[0, 1, 2]
```

```
[9.9, 8.8, 7.7]
```

وفي حال أردنا الوصول إلى العناصر الموجودة في القوائم الداخلية، فيمكننا استعمال حلقة for متشعبة:

```
list_of_lists = [['hammerhead', 'great white', 'dogfish'],[0, 1, 2],[9.9, 8.8, 7.7]]

for list in list_of_lists:
   for item in list:
     print(item)
```

الناتج:

```
hammerhead
great white
dogfish
0
1
2
9.9
8.8
7.7
```

نستطيع الاستفادة من حلقات for المتشعبة عندما نريد الدوران على عناصر محتوى في قوائم.

3. التحكم بحلقات التكرار

وجدنا أنَّ استخدام حلقات for أو while تسمح بأتمتة وتكرار المهام بطريقة فعّالة. لكن في بعض الأحيان، قد يتدخل عامل خارجي في طريقة تشغيل برنامجك، وعندما يحدث ذلك،

فربما تريد من برنامجك الخروج تمامًا من حلقة التكرار، أو تجاوز جزء من الحلقة قبل إكمال تنفيذها، أو تجاهل هذا العامل الخارجي تمامًا. لذا يمكنك فعل ما سبق باستخدام التعليمات break و continue و pass.

أ. التعليمة break

توفِّر لك التعليمة break القدرة على الخروج من حلقة التكرار عند حدوث عامل خارجي. فعليك وضع التعليمة break في الشيفرة التي ستُنفَّذ في كل تكرار للحلقة، وتوضع عادةً ضمن تعليمة شرطية مثل if. ألقِ نظرةً إلى أحد الأمثلة الذي يستعمل التعليمة break داخل حلقة for:

```
number = 0

for number in range(10):
  number = number + 1 (*)

  if number == 5:
    break    # توقف هنا

  print('Number is ' + str(number))

print('Out of loop')
```

هذا برنامجٌ صغيرٌ، هيّأنا في بدايته المتغير number بجعله يساوي الصفر، ثم بنينا حلقة تكرار for التي تعمل لطالما كانت قيمة المتغير number أصغر من 10. ثم قمنا بزيادة قيمة المتغير number داخل حلقة for بمقدار 1 في كل تكرار، وذلك في السطر (*) ثم كان هنالك الشرط if الذي يختبر إن كان المتغير number مساوٍ للرقم 5، وعند حدوث ذلك فستُنفَّذ التعليمة

break للخروج من الحلقة، وتوجد داخل حلقة التكرار الدالة print() التي تُنفَّذ في كل تكرار إلى أن نخرج من الحلقة عبر التعليمة break، إذ هي موجودة بعد التعليمة break. لكي نتأكد أننا خرجنا من الحلقة، وضعنا عبارة print() أخيرة موجودة خارجها. سنرى الناتج الآتي عند تنفيذ البرنامج:

```
Number is 1
Number is 2
Number is 3
Number is 4
Out of loop
```

يُظهِر الناتج السابق أنَّه بمجرد أن أصبح العدد الصحيح number مساويًا للرقم 5، فسينتهي تنفيذ حلقة التكرار عبر break.

ب. التعليمة continue

تسمح لنا التعليمة continue بتخطي جزء من حلقة التكرار عند حدوث عامل خارجي، وعدم إكمال بقية الحلقة إلى نهايتها. بعبارةٍ أخرى، سينتقل تنفيذ البرنامج إلى أوّل حلقة التكرار عند تنفيذ التعليمة continue. يجب وضع التعليمة continue في الشيفرة التي ستُنفَّذ في كل تكرار للحلقة، ويوضع عادةً ضمن الشرط if.

سنستخدم نفس البرنامج الذي استعملناها لشرح التعليمة break أعلاه، لكننا سنستخدم التعليمة continue بدلًا من break:

```
number = 0

for number in range(10):
  number = number + 1
```

```
    if number == 5:
        continue    # continue here

    print('Number is ' + str(number))

print('Out of loop')
```

الفرق بين استخدام التعليمة `continue` بدلًا من `break` هو إكمال تنفيذ الشيفرة بغض النظر عن التوقف الذي حدث عندما كانت قيمة المتغير `number` مساويةً إلى الرقم 5. لننظر إلى الناتج:

```
Number is 1
Number is 2
Number is 3
Number is 4
Number is 6
Number is 7
Number is 8
Number is 9
Number is 10
Out of loop
```

نلاحظ أنَّ السطر الذي يجب أن يحتوي على `Number is 5` ليس موجودًا في المخرجات، لكن سيُكمَل تنفيذ حلقة التكرار بعد هذه المرحلة مما يطبع الأرقام من 6 إلى 10 قبل إنهاء تنفيذ الحلقة.

يمكنك استخدام التعليمة `continue` لتفادي استخدام تعليمات شرطية معقَّدة ومتشعِّبة، أو لتحسين أداء البرنامج عن طريق تجاهل الحالات التي ستُرفَض نتائجها.

ج. التعليمة pass

تسمح لنا التعليمة pass بالتعامل مع أحد الشروط دون إيقاف عمل حلقة التكرار بأي شكل، أي ستُنفَّذ جميع التعليمات البرمجية الموجودة في حلقة التكرار ما لم تستعمل تعليمات تحكم مثل break أو continue فيها. وكما هو الحال مع التعليمات السابقة، يجب وضع التعليمة pass في الشيفرة التي ستُنفَّذ في كل تكرار للحلقة، ويوضع عادةً ضمن الشرط if. سنستخدم نفس البرنامج الذي استعملناها لشرح التعليمة break أو continue أعلاه، لكننا سنستخدم التعليمة pass هذه المرة:

```
number = 0

for number in range(10):
  number = number + 1

  if number == 5:
    pass   # تخطى وأكمل

  print('Number is ' + str(number))

print('Out of loop')
```

تخبر التعليمة pass التي تقع بعد الشرط if البرنامج أنَّ عليه إكمال تنفيذ الحلقة وتجاهل مساواة المتغير number للرقم 5. لنشغِّل البرنامج ولننظر إلى الناتج:

```
Number is 1
Number is 2
Number is 3
Number is 4
```

```
Number is 5
Number is 6
Number is 7
Number is 8
Number is 9
Number is 10
Out of loop
```

لاحظنا عند استخدامنا للتعليمة pass في هذا البرنامج أنَّ البرنامج يعمل كما لو أننا لم نضع تعليمة شرطية داخل حلقة التكرار؛ إذ تخبر التعليمة pass البرنامج أن يكمل التنفيذ كما لو أنَّ الشرط لم يتحقق. يمكن أن تستفيد من التعليمة pass عندما تكتب برنامجك لأوّل مرة أثناء تفكيرك بحلّ مشكلة ما عبر خوارزمية، لكن قبل أن تضع التفاصيل التقنية له.

4. خلاصة الفصل

شرحنا في هذا الفصل كيف تعمل حلقتي التكرار while ، و for في بايثون وكيفية إنشائها، إذ تستمر الأولى بتنفيذ مجموعة من الأسطر البرمجية لطالما كان الشرط مساويًا للقيمة true بينما تستمر الثانية بتنفيذ مجموعة من الشيفرات لعددٍ مُحدّدٍ من المرات. انتقلنا أخيرًا إلى التعليمات break و continue و pass التي تسمح بالتحكم أكثر بحلقات for و while والتعامل مع الأحداث المفاجئة التي تحدث أثناء تنفيذ مهمة مُتكرّرة.

16

الدوال: تعريفها واستعمالها

الدالة (function) هي كتلة من التعليمات التي تنفِّذ إجراءً ما، ويمكن، بعد تعريفها، إعادة استخدامها في أكثر من موضع. تجعل الدوال الشيفرة تركيبية (modular)، مما يسمح باستخدام نفس الشفرة مرارًا وتكرارًا.

تضم بايثون عددًا من الدوال المُضمّنة الشائعة، مثل:

- `print()` والتي تطبع كائنًا في الطرفية،
- `int()` والتي تحوِّل أنواع البيانات النصية أو العددية إلى أعداد صحيحة،
- `len()` التي تعيد طول كائن، وغيرها من الدوال.

سنتعلَّم في هذا الفصل كيفيَّة تعريف الدوال، وكيفيَّة استخدامها في البرامج.

1. تعريف دالة

لنبدأ بتحويل البرنامج الذي يطبع عبارة "مرحبًا بالعالم!" إلى دالة.

أنشئ ملفًا نصيًا جديدًا، وافتحه في محرر النصوص المفضل عندك، ثم استدع البرنامج `hello.py`. تُعرَّف الدالة باستخدام الكلمة المفتاحية `def`، متبوعة باسم من اختيارك، متبوعًا بقوسين يمكن أن يَحتويا المعاملات التي ستأخذها الدالة، ثم ينتهي التعريف بنقطتين. في هذه الحالة، سنعرِّف دالة باسم `hello()`:

```
def hello():
```

في الشفرة أعلاه، أعددنا السطر الأول من تعريف الدالة.

بعد هذا، سنضيف سطرًا ثانيًا مُزاحًا بأربع مسافات بيضاء، وفيه سنكتب التعليمات التي ستنفِّذها الدالة. في هذه الحالة، سنطبع العبارة "مرحبًا بالعالم!" في سطر الأوامر:

```
def hello():
    print("مرحبا بالعالم")
```

لقد أتممنا تعريف دالتنا، غير أننا إن نَفَّذنا البرنامج الآن، فلن يحدث أيُّ شيء، لأننا لم نستدع الدالة؛ لذلك، سنستدعي الدالة بالشكل `hello()` خارج كتلة تعريف الدالة:

```
def hello():
    print("مرحبا بالعالم!")

hello()
```

الآن، لننفِّذ البرنامج:

```
python hello.py
```

يجب أن تحصل على المخرجات التالية:

```
مرحبا بالعالم!
```

بعض الدوال أكثر تعقيدًا بكثير من الدالة `hello()` التي عرَّفناها أعلاه. على سبيل المثال، يمكننا استخدام الحلقة `for` والتعليمات الشرطية، وغيرها داخل كتلة الدالة.

على سبيل المثال، تستخدم الدالة المُعرَّفة أدناه تعليمة شرطية للتحقُّق مما إذا كانت المدخلات الممرَّرة إلى المتغير `name` تحتوي على حرف علة (vowel)، ثم تستخدم الحلقة `for` للمرور (iterate) على الحروف الموجودة في السلسلة النصية `name`.

```
# names() تعريف الدالة
def names():
    # إعداد المتغير name وإحالة المدخلات عليه
    name = str(input('أدخل اسمك باللغة الإنجليزية:'))
```

```
    # التحقق من أن name يحتوي حرف علة
    if set('aeiou').intersection(name.lower()):
      print('اسمك يحوي حرف علة')
    else:
      print('اسمك لا يحوي حرف علة')

    # المرور على حروف name
    for letter in name:
      print(letter)

# استدعاء الدالة
names()
```

تستخدم الدالة `names()` التي عرَّفناها أعلاه تعليمة شرطية، وحلقة `for`، وهذا توضيح لكيفية تنظيم الشفرة البرمجية ضمن تعريف الدالة. يمكننا أيضًا جعل التعليمة الشرطية والحلقة `for` دالتين منفصلتين.

تعريف الدوال داخل البرامج يجعل الشفرة البرمجية تركيبية (modular)، وقابلة لإعادة الاستخدام، وذلك سيتيح لنا استدعاء نفس الدالة دون إعادة كتابة شيفرتها كل مرة.

2. المعاملات: تمرير بيانات للدوال

حتى الآن، عرَّفنا دالة ذات قوسين فارغين لا تأخذ أيَّ وسائط (arguments)، سنتعلم في هذا القسم كيفية تعريف المعاملات (parameters) وتمرير البيانات إلى الدوال.

المعامل (parameter) هو كيان مُسمًّى يوضع في تعريف الدالة، ويعرِّف وسيطًا (arguments) يمكن أن تقبله الدالة عند استدعائها.

دعنا ننشئ برنامجًا صغيرًا يأخذ ثلاثة معاملات x و y و z. سننشئ دالة تجمع تلك المعاملات وفق عدة مجموعات ثمَّ تطبع حاصل جمعها.

```
def add_numbers(x, y, z):
    a = x + y
    b = x + z
    c = y + z
    print(a, b, c)

add_numbers(1, 2, 3)
```

مرّرنا العدد 1 إلى المعامل x، و 2 إلى المعامل y، و 3 إلى المعامل z. تتوافق هذه القيم مع المعاملات المقابلة لها في ترتيب الظهور.

يُجرِي البرنامج العمليات الحسابية على المعاملات على النحو التالي:

```
a = 1 + 2
b = 1 + 3
c = 2 + 3
```

تطبع الدالة أيضًا a و b و c، وبناءً على العمليات الحسابية أعلاه، فإنَّ قيمة a ستساوي العدد 3، و b ستساوي 4، و c ستساوي العدد 5.

لننفِّذ البرنامج:

```
python add_numbers.py
```

سنحصل على المخرجات التالية:

```
3 4 5
```

المعاملات هي متغيرات تُعرَّف ضمن جسم الدالة. يمكن تعيين قيم إليها عند تنفيذ التابع بتمرير وسائط إلى الدالة، إذ تُسنَد الوسائط إليها تلقائيًا آنذاك.

3. الوسائط المسمَّاة

تُستدعى المعاملات بحسب ترتيب ظهورها في تعريف الدالة، أما الوسائط المسماة (Keyword Arguments) فتُستخدَم بأسمائها في استدعاء الدالة.

عند استخدام الوسائط المسمّاة، يمكنك استخدام المعاملات بأيِّ ترتيب تريد، لأنَّ مترجم بايثون سيستخدم الكلمات المفتاحية لمطابقة القيم مع المعاملات.

سننشئ دالة تعرض معلومات الملف الشخصي للمستخدم، ونمرِّر إليها المُعامِلين username (سلسلة نصية)، و followers (عدد صحيح).

```
# تعريف دالة ذات معاملات
def profile_info(username, followers):
    print("Username: " + username)
    print("Followers: " + str(followers))
```

داخل تعريف الدالة، وضعنا username و followers بين قوسي الدالة profile_info() أثناء تعريفها. تطبع شفرة الدالة المعلومات الخاصَّة بالمستخدم على هيئة سلسلة نصية باستخدام المعاملين المُمرّرين.

الآن، يمكننا استدعاء الدالة وتعيين المعاملات:

```
def profile_info(username, followers):
    print("Username: " + username)
    print("Followers: " + str(followers))

# استدعاء الدالة مع تعيين المعاملات
profile_info("sammyshark", 945)

# استدعاء الدالة مع تمرير الوسائط المسماة إليها
profile_info(username="AlexAnglerfish", followers=342)
```

في الاستدعاء الأول للدالة، مرَّرنا اسم المستخدم sammyshark، وعدد المتابعين 945 بالترتيب الوارد في تعريف الدالة. أمَّا في الاستدعاء الثاني للدالة، فقد استخدمنا الوسائط المسمَّاة، وعيَّنا قيمًا للوسائط ويمكن عكس الترتيب إن شئنا. لننفذ البرنامج:

```
python profile.py
```

سنحصل على المخرجات التالية:

```
Username: sammyshark
Followers: 945
Username: AlexAnglerfish
Followers: 342
```

سنحصل في المخرجات على أسماء المستخدمين، وأعداد المتابعين لكلا المستخدمين.

يمكننا تغيير ترتيب المعاملات، كما في المثال التالي:

```
def profile_info(username, followers):
    print("Username: " + username)
    print("Followers: " + str(followers))

# تغيير ترتيب المعاملات
profile_info(followers=820, username="cameron-catfish")
```

عند تنفيذ البرنامج أعلاه، سنحصل على المخرجات التالية:

```
Username: cameron-catfish
Followers: 820
```

يحافظ تعريف الدالة على نفس ترتيب التعليمات في print()، لذلك يمكننا استخدام الوسائط المسمّاة بأيِّ ترتيب نشاء.

4. القيم الافتراضية للوسائط

يمكننا إعطاء قيم افتراضية لواحد أو أكثر من المعاملات. في المثال أدناه، سنعطي للمعامل followers القيمة الافتراضية 1 لاستعمالها إن لم تُمرَّر هذه القيمة للدالة عند استدعائها:

```
def profile_info(username, followers=1):
    print("Username: " + username)
    print("Followers: " + str(followers))
```

الآن، يمكننا استدعاء الدالة مع تعيين اسم المستخدم فقط، وسيُعيَّن عدد المتابعين تلقائيًا ويأخذ القيمة 1. لكن يمكننا تغيير عدد المتابعين إن شئنا.

```
def profile_info(username, followers=1):
    print("Username: " + username)
    print("Followers: " + str(followers))

profile_info(username="JOctopus")
profile_info(username="sammyshark", followers=945)
```

عندما ننفِّذ البرنامج باستخدام الأمر python profile.py، ستظهر المخرجات التالية:

```
Username: JOctopus
Followers: 1
Username: sammyshark
Followers: 945
```

تمرير قيم إلى المعاملات الافتراضية سيتخطى القيمة الافتراضية المعطاة في تعريف الدالة.

5. إعادة قيمة

كما يمكن تمرير قيم إلى الدالة، فيمكن كذلك أن تنتج الدالة قيمة وتعيدها لمن استدعاها.

يمكن أن تنتج الدالة قيمة، ويكونُ ذلك عبر استخدام التعليمة `return`؛ هذه التعليمة اختيارية، وفي حال استخدامها، فستُنهِي الدالة مباشرةً عملها وتوقف تنفيذها، وتُمرَّر قيمة التعبير الذي يعقُبها إلى المُستدعي (caller). إذا لم يلي التعليمة `return` أي شيء، فستُعيد الدالة القيمةَ `None`.

حتى الآن، استخدمنا الدالة `print()` بدلاً من `return` في دوالنا لطباعة شيء بدلًا من إعادته. لننشئ برنامجًا يعيد متغيرًا بدلًا من طباعته الآن.

سننشئ برنامجًا في ملف نصي جديد يسمى `square.py` يحسب مربع المعامل `x`، ويُحيل الناتج إلى المتغير `y`، ثم يعيده. سنطبع المتغير `result`، والذي يساوي ناتج تنفيذ الدالة `square(3)`.

```
def square(x):
    y = x ** 2
    return y

result = square(3)
print(result)
```

لننفّذ البرنامج:

```
python square.py
```

سنحصل على المخرجات التالية:

```
9
```

مخرجات البرنامج هي العدد الصحيح 9 الذي أعادته الدالة وهو ما نتوقعه لو طلبنا من بايثون حساب مربع العدد 3.

لفهم كيفية عمل التعليمة `return`، يمكننا تعليق التعليمة `return`:

```
def square(x):
    y = x ** 2
    # return y

result = square(3)
print(result)
```

الآن، لننفّذ البرنامج مرة أخرى:

```
python square.py
```

سنحصل على الناتج التالي:

```
None
```

بدون استخدام التعليمة `return`، لا يمكن للبرنامج إعادة أيِّ قيمة، لذلك تُعاد القيمة الافتراضية `None`.

إليك مثال آخر، في برنامج `add_numbers.py` أعلاه، سنستبدل بالتعليمة `return` الدالة `print()`:

```
def add_numbers(x, y, z):
    a = x + y
    b = x + z
    c = y + z
    return a, b, c
```

```
sums = add_numbers(1, 2, 3)
print(sums)
```

خارج الدالة، أحلنا إلى المتغير sums نتيجة استدعاء الدالة بالوسائط 1 و 2 و 3 كما فعلنا أعلاه ثم طبعنا قيمته.

فلننفِّذ البرنامج مرة أخرى:

```
python add_numbers.py
```

والناتج سيكون:

```
(3, 4, 5)
```

لقد حصلنا على الأعداد 3 و 4 و 5 وهي نفس المخرجات التي تلقيناها سابقًا عندما استخدمنا الدالة print() في الدالة. هذه المرة تمت إعادتها على هيئة صف لأنَّ التعبير المرافق للتعليمة return يحتوي على فاصلة واحدة على الأقل.

تُوقَف الدوال فورًا عندما تصل إلى التعليمة return، سواء أعادت قيمة، أم لم تُعِد.

```
def loop_five():
    for x in range(0, 25):
        print(x)
        if x == 5:
            # x == 5 إيقاف الدالة عند
            return
    print("This line will not execute.")

loop_five()
```

يـؤدي اسـتخدام التعليمـة return داخـل الحلقـة for إلى إنهـاء الدالـة، وبالتـالي لن يُنفَّذ السطر الموجود خارج الحلقة.

لو استخدمنا بدلًا من ذلك break، فسيُنفَّذ السطر print() الأخير من المثال السابق. نعيد التذكير أنَّ التعليمة return تنهي عمل الدالة، وقد تعيد قيمة إذا أعقبها تعبير.

6. استخدام main() دالةً رئيسيةً

رغم أنَّه يمكنـك في بـايثون اسـتدعاء الدالـة في أسـفل البرنـامج وتنفيـذها (كمـا فعلنـا في الأمثلـة أعلاه)، فـإنَّ العديـد من لغـات البرمجـة (مثـل C++ و Java) تتطلب وجـود دالـة رئيسـية تدعى main. إنَّ تضمين دالة main()، وإن لم يكن إلزاميًا، يمكن أن يهيكل برامج بـايثون بطريقة منطقية، فتضع أهم مكونات البرنامج في دالة واحدة. كما يمكن أن يجعل البرنـامج أكثر مقروئيـة للمبرمجين غير البايثونيِّين.

سـنبدأ بإضـافة دالة main() إلى برنـامج hello.py أعلاه. سـنحتفظ بالدالة hello()، ثم نعرِّف الدالة main():

```
def hello():
    print("مرحبا بالعالم")

def main():
```

ضمن الدالة main()، سندرج الدالة print()، والتي ستعُلِمنا بأنَّنا في الدالة main(). أيضًا سنستدعي الدالة hello() داخل main():

```
def hello():
    print("مرحبا بالعالم")

def main():
    print("هذه هي الدالة الرئيسية")
    hello()
```

أخيرًا، في أسفل البرنامج، سنستدعي الدالة `main()`:

```
def hello():
    print("مرحبا بالعالم!")

def main():
    print("هذه هي الدالة الرئيسية.")
    hello()

main()
```

الآن يمكننا تنفيذ برنامجنا:

```
python hello.py
```

وسنحصل على المخرجات التالية:

```
هذه هي الدالة الرئيسية.
مرحبا بالعالم!
```

لمَّا استدعينا الدالة `hello()` داخل `main()`، ثم نفَّذنا الدالة `main()` وحدها، فقد طُبع النص "مرحبا بالعالم!" مرة واحدة فقط، وذلك عقب السلسلة النصية التي أخبرتنا بأننا في الدالة الرئيسية. (سنعمل الآن مع دوال مُتعدِّدة، لذلك من المستحسن أن تراجع نطاقات المتغيرات في فصل المتغيرات إن كنت قد نسيتها.)

إذا عرَّفت متغيرًا داخل دالة، فلا يمكنك أن تستخدم ذلك المتغير إلا ضمن تلك الدالة. لذا، إن أردت استخدام متغير ما في عدة دوال، فقد يكون من الأفضل الإعلان عنه متغيرًا عامًا (global variable).

في بايثون، يعدُّ `'__main__'` اسم النطاق الذي ستُنفَّذ فيه الشيفرة العليا (top-level code). عند تنفيذ برنامج من الدخل القياسي (standard input)، أو من سكربت، أو من سطر الأوامر، سيتم ضبط `__name__` عند القيمة `'__main__'`.

لهذا السبب، اصطلح مطورو بايثون على استخدام الصياغة التالية:

```
if __name__ == '__main__':
    # الشفرة التي ستُنفّذ لو كان هذا هو البرنامج الرئيسي
```

هذه الصياغة تتيح استخدام ملفات بايثون إما:

- برامج رئيسية، مع تنفيذ ما يلي التعليمة `if`، أو
- وحدات عادية، مع عدم تنفيذ ما يتبع التعليمة `if`.

ستُنفَّذ الشفرة غير المُتضمِّنة في العبارة `if __name__ == '__main__':` عند التنفيذ. إذا كنت تستخدم ملف بايثون كوحدة، فستُنفَّذ أيضًا الشيفرة البرمجية غير المُتضمّنة في هذه العبارة عند استيراد ذلك الملف.

دعنا نوسّع البرنامج `names.py` أعلاه، لننشئ ملفا جديدًا يسمى `more_names.py`. سنعلن في هذا البرنامج عن متغير عام، ونعدِّل الدالة `names()` الأصليَّة بشكل نقسِّم فيه التعليمات إلى دالتين منفصلتين.

ستتحقّق الدالة الأولى has_vowel() مما إذا كانت السلسلة النصية name تحتوي على حرف علة (vowel). وتطبع الدالة الثانية print_letters() كل حرف من السلسلة النصية name.

```
# الإعلان عن متغير عام لاستخدامه في جميع الدوال
name = str(input('أدخل اسمك باللغة الإنجليزية:'))


# تعريف دالة للتحقق من أن name يحتوي حرف علة
def has_vowel():
    if set('aeiou').intersection(name.lower()):
        print('اسمك يحتوي حرف علة')
    else:
        print('اسمك لا يحتوي حرف علة')


# المرور على حروف name
def print_letters():
    for letter in name:
        print(letter)
```

بعد ذلك، دعنا نعرِّف الدالة main() التي سَتستدعي كلا من الدالة has_vowel() والدالة print_letters().

```
# الإعلان عن متغير عام لاستخدامه في جميع الدوال
name = str(input('أدخل اسمك باللغة الإنجليزية:'))


# تعريف دالة للتحقق من أنّ name يحتوي حرف علة
def has_vowel():
    if set('aeiou').intersection(name.lower()):
```

```
        print('اسمك يحتوي حرف علة')
    else:
        print('اسمك لا يحتوي حرف علة')


# المرور على حروف name
def print_letters():
    for letter in name:
        print(letter)


# تعريف الدالة main التي ستستدعي بقية الدوال
def main():
    has_vowel()
    print_letters()
```

أخيرًا، سنضيف العبارة `if __name__ == '__main__':` في أسفل الملف. لقد وضعنا جميع الدوال التي نودُّ تنفيذها في الدالة `main()`، لذا سنستدعي الدالة `main()` بعد الشرط `if`.

```
# الإعلان عن متغير عام لاستخدامه في جميع الدوال
name = str(input('أدخل اسمك:'))


# تعريف دالة للتحقق من أن name يحتوي حرف علة
def has_vowel():
    if set('aeiou').intersection(name.lower()):
        print('اسمك يحتوي حرف علة')
    else:
        print('اسمك لا يحتوي حرف علة')


# المرور على حروف name
```

```
def print_letters():
    for letter in name:
        print(letter)

# تعريف الدالة main التي ستستدعي بقية الدوال

def main():
    has_vowel()
    print_letters()

# تنفيذ الدالة main()
if __name__ == '__main__':
    main()
```

يمكننا الآن تنفيذ البرنامج:

```
python more_names.py
```

سيعرض هذا البرنامج نفس المخرجات التي عرضها البرنامج `names.py`، بيْد أنَّ الشفرة هنا أكثر تنظيمًا، ويمكن استخدامها بطريقة تركيبية (modular).

إذا لم ترغب في الإعلان عن الدالة `main()`، يمكنك بدلًا من ذلك إنهاء البرنامج كما يلي:

```
...
if __name__ == '__main__':
    has_vowel()
    print_letters()
```

يؤدي استخدام دالة رئيسية مثل `main()`، واستخدام العبارة `if __name__ == '__main__':` إلى تنظيم الشيفرة البرمجية بطريقة منطقية، وجعلها أكثر مقروئية وتراكيبية.

7. استخدام args* و kwargs**

تعدُّ المعاملات في تعاريف الدوال كيانات مسماة تُحدِّد وسيطًا (argument) يمكن أن يُمرَّر إلى الدالة المُعرَّفة.

أثناء البرمجة، قد لا تدرك جميع حالات الاستخدام الممكنة للشيفرة، لذا قد ترغب في توسيع خيارات المبرمجين المستقبليين الذين سيستخدمون الوحدة التي طورتها.

سنتعلم في هذا القسم كيفيَّة تمرير عدد متغير من الوسائط إلى دالة ما باستخدام الصياغتين `*args` و `**kwargs`.

أ. `*args`

في بايثون، يمكن استخدام الشكل أحادي النجمة `*args` معاملًا لتمرير قائمة غير محدَّدة الطول من الوسائط غير المسماة (non-keyworded argument) إلى الدوال. تجدر الإشارة إلى أنَّ النجمة (*) عنصر ضروري هنا، إذ رغم أنَّ الكلمة `args` متعارف عليها بين المبرمجين، إلا أنَّها غير رسمية.

لنلقِ نظرة على مثال لدالة تستخدم وسيطين:

```
def multiply(x, y):
    print (x * y)
```

عرَّفنا في الشيفرة أعلاه دالة تقبل وسيطين `x` و `y`، عندما نستدعي هذه الدالة، سنحتاج إلى تمرير عددين موافقين للوسيطين `x` و `y`. في هذا المثال، سنمرِّر العدد الصحيح `5` إلى `x`، والعدد الصحيح `4` إلى `y`:

```
def multiply(x, y):
   print (x * y)

multiply(5, 4)
```

عند تنفيذ الشفرة أعلاه:

```
python lets_multiply.py
```

سنحصل على المخرجات التالية:

```
20
```

ماذا لو قرّرنا لاحقًا حساب ناتج ضرب ثلاثة أعداد بدلًا من عددين فقط؟ إذا حاولت تمرير عدد إضافي إلى الدالة، كما هو موضح أدناه:

```
def multiply(x, y):
   print (x * y)

multiply(5, 4, 3)
```

فسيُطلَق الخطأ التالي:

```
TypeError: multiply() takes 2 positional arguments but 3 were
given
```

إذا شككت أنّك ستحتاج إلى استخدام المزيد من الوسائط لاحقًا، فالحل هو استخدام `*args` معاملا. سنزيل المُعاملين `x` و `y` من الشيفرة في المثال الأول، ونضع مكانهما `*args`:

```
def multiply(*args):
    z = 1
    for num in args:
        z *= num
    print(z)


multiply(4, 5)
multiply(10, 9)
multiply(2, 3, 4)
multiply(3, 5, 10, 6)
```

عندما ننفِّذ هذه الشيفرة، سنحصل على ناتج استدعاءات الدالة أعلاه:

```
20
90
24
900
```

يمكننا باستخدام الوسيط `*args` تمرير أي عدد نحب من الوسائط عند استدعاء الدالة بالإضافة إلى كتابة شيفرة أكثر مرونة، وإنشاء دوال تقبل عددًا غير محدد مسبقًا من الوسائط غير المسماة.

ب. `**kwargs`

يُستخدَم الشكل ذو النجمتين `**kwargs` لتمرير قاموس متغيَّر الطول من الوسائط المسماة إلى الدالة المعرَّفة. مرة أخرى، النجمتان (`**`) ضروريتان، فمع أنَّ استخدام الكلمة `kwargs` متعارف عليه لدى المبرمجين، إلا أنَّها غير رسمية.

يمكن أن تأخذ `**kwargs`، كما هو شأن `*args`، أيَّ عدد من الوسائط التي ترغب في

تمريرها إلى الدالة بيْد أنّ `**kwargs` تختلف عن `*args` في أنَّها تستوجب تعيين أسماء المعاملات (keywords).

في المثال التالي، ستطبع الدالة الوسيط `**kwargs` الممرر إليها:

```
def print_kwargs(**kwargs):
        print(kwargs)
```

سنستدعي الآن الدالة ونمرِّر إليها بعض الوسائط المسماة:

```
def print_kwargs(**kwargs):
        print(kwargs)

print_kwargs(kwargs_1="Shark", kwargs_2=4.5, kwargs_3=True)
```

لننفِّذ البرنامج أعلاه:

```
python print_kwargs.py
```

سنحصل على المخرجات التالية:

```
{'kwargs_3': True, 'kwargs_2': 4.5, 'kwargs_1': 'Shark'}
```

اعتمادًا على إصدار بايثون 3 الذي تستخدمه، فقد لا يكون القاموس مرتبًا. في بايثون 3.6 وما بعده، ستحصل على أزواج قيمة-مفتاح (key-value) مرتبة، ولكن في الإصدارات السابقة، سيكون ترتيب الأزواج عشوائيًا.

سيُنشَأ قاموس يسمى `kwargs`، والذي يمكننا التعامل معه مثل أي قاموس عادي داخل الدالة.

لننشئ برنامجًا آخر لإظهار كيفيَّة استخدام `**kwargs`. سننشئ دالة تطبع قاموسًا من الأسماء. أولاً، سنبدأ بقاموس يحوي اسمين:

```
def print_values(**kwargs):
    for key, value in kwargs.items():
        print("The value of {} is {}".format(key, value))

print_values(my_name="Sammy", your_name="Casey")
```

بعد تنفيذ البرنامج عبر الأمر التالي:

```
python print_values.py
```

سنحصل على ما يلي:

```
The value of your_name is Casey
The value of my_name is Sammy
```

قد لا تكون القواميس مرتَّبة، لذلك قد يظهر الاسم `Casey` أولًا، وقد يظهر ثانيًا.

سنُمرِّر الآن وسائط إضافيَّة إلى الدالة لنرى كيف يمكن أن تقبل `**kwargs` أيَّ عدد من الوسائط:

```
def print_values(**kwargs):
    for key, value in kwargs.items():
        print("The value of {} is {}".format(key, value))

print_values(
            name_1="Alex",
            name_2="Gray",
            name_3="Harper",
            name_4="Phoenix",
            name_5="Remy",
```

```
            name_6="Val"
        )
```

إذا نفَّذنا البرنامج الآن، فسنحصل على المخرجات التالية، والتي قد تكون غير مرتبة:

```
The value of name_2 is Gray
The value of name_6 is Val
The value of name_4 is Phoenix
The value of name_5 is Remy
The value of name_3 is Harper
The value of name_1 is Alex
```

يتيح لك استخدام `**kwargs` مرونةً كبيرةً في استخدام الوسائط المسماة. فعند استخدامها، لن نحتاج إلى معرفة مسبقة بعدد الوسائط التي ستمرر إلى الدالة.

8. ترتيب الوسائط

عند الخلط بين عدة أنواع من الوسائط داخل دالة، أو داخل استدعاء دالة، يجب أن تظهر الوسائط وفق الترتيب التالي:

- الوسائط العادية
- `*args`
- الوسائط المسمّاة
- `**kwargs`

عمليًا، عند الجمع بين المعاملات العادية، والوسيطين `*args` و `**kwargs`، فينبغي أن تكون وفق الترتيب التالي:

```
def example(arg_1, arg_2, *args, **kwargs):
...
```

وعند الجمع بين المعاملات العادية والمعاملات المسماة و *args و **kwargs، ينبغي أن تكون وفق الترتيب التالي:

```
def example2(arg_1, arg_2, *args, kw_1="shark",
kw_2="blobfish", **kwargs):
...
```

من المهم أن تأخذ في الحسبان ترتيب الوسائط عند إنشاء الدوال حتى لا تتسبب في إطلاق خطأٍ متعلقٍ بالصياغة.

9. استخدام *args و **kwargs في استدعاءات الدوال

يمكننا أيضًا استخدام *args و **kwargs لتمرير الوسائط إلى الدوال. أولًا، دعنا ننظر إلى مثال يستخدم *args:

```
def some_args(arg_1, arg_2, arg_3):
    print("arg_1:", arg_1)
    print("arg_2:", arg_2)
    print("arg_3:", arg_3)

args = ("Sammy", "Casey", "Alex")
some_args(*args)
```

في الدالة أعلاه، هناك ثلاثة معاملات، وهي arg_1 و arg_ و arg_3. ستطبع الدالة كل هذه الوسائط. بعد ذلك أنشأنا متغيرًا، وأَحلنا عليه عنصرًا تكراريًا (في هذه الحالة، صف)، ثم مرَّرنا ذلك المتغير إلى الدالة باستخدام الصياغة النجمية (asterisk syntax).

عنـدما ننفِّذ البرنـامج باسـتخدام الأمـر python some_args.py، سنحصـل على المخرجات التالية:

```
arg_1: Sammy
arg_2: Casey
arg_3: Alex
```

يمكننـا أيضًـا تعـديل البرنـامج أعلاه، واسـتخدام قائمـة. سـندمج أيضًـا ‎*args مـع وسيط مسمى:

```
def some_args(arg_1, arg_2, arg_3):
    print("arg_1:", arg_1)
    print("arg_2:", arg_2)
    print("arg_3:", arg_3)

my_list = [2, 3]
some_args(1, *my_list)
```

إذا نفَّذنا البرنامج أعلاه، فسنحصل على المخرجات التالية:

```
arg_1: 1
arg_2: 2
arg_3: 3
```

وبالمثل، يمكن استخدام الوسائط المسماة ‎**kwargs لاستدعاء دالة.

سننشـئ متغيـرًا، ونسـند إليـه قاموسًـا من 3 أزواج مفتـاح-قيمـة (سنسـتخدم kwargs هنا، ولكن يمكنك تسميته ما تشاء)، ثم نُمرِّره إلى دالة ذات 3 وسائط:

```
def some_kwargs(kwarg_1, kwarg_2, kwarg_3):
    print("kwarg_1:", kwarg_1)
    print("kwarg_2:", kwarg_2)
    print("kwarg_3:", kwarg_3)

kwargs = {"kwarg_1": "Val", "kwarg_2": "Harper", "kwarg_3":
"Remy"}
some_kwargs(**kwargs)
```

عند تنفيذ البرنامج أعلاه باستخدام الأمر `python some_kwargs.py`، سنحصل على المخرجات التالية:

```
kwarg_1: Val
kwarg_2: Harper
kwarg_3: Remy
```

10. خلاصة الفصل

الدوال هي كتل من التعليمات البرمجية التي تُنفِّذ إجراءات معيَّنة داخل البرنامج، كما تساعد على جعل الشيفرة تركيبية، وقابلة لإعادة الاستخدام بالإضافة إلى تنظيمها وتسهيل قراءتها. (لمعرفة المزيد حول كيفية جعل الشفرة تركيبية، يمكنك الرجوع إلى فصل الوحدات.)

يمكنك استخدام الصياغتين `*args` و `**kwargs` الخاصتين ضمن تَعاريف وسائط الدوال لتمرير أيِّ عدد تشاء من الوسائط إليها، إذ يُستحسن استخدام `*args` و `**kwargs` في المواقف التي تتوقع أن يظل فيها عدد المدخلات في قائمة الوسائط صغيرًا نسبيًا. وضع في ذهنك أن استخدام `*args` و `**kwargs` يحسِّن مقروئية الشيفرة، ويسهِّل على المبرمجين، ولكن ينبغي استخدامهما بحذر.

17

الوحدات: استيرادها وإنشاؤها

توفر لغة بايثون مجموعة متنوعة من الدوال المضمَّنة مثل:

- `print()`: تطبع التعابير المُمرَّرة إليها في مجرى الخرج
- `abs()`: تُعيد القيمة المطلقة للعدد
- `int()`: تحوِّل القيمة المُمرَّرة إليها إلى عدد صحيح
- `len()`: تُعيد طول تسلسل أو مجموعة

هذه الدوال المضمَّنة مفيدة، لكنها محدودة، لهذا يستخدم المطورون الوحدات (modules) لتطوير برامج أكثر تعقيدًا. الوحدات (Modules) هي ملفات بايثون ذات امتداد `.py`، والتي تحوي شيفرات بايثون. يمكن التعامل مع أيِّ ملف بايثون على أنه وحدة. مثلًا، إن كان هناك ملف بايثون يسمى `hello.py`، فسيكون اسم الوحدة المقابلة له `hello`، والذي يمكن استيراده في ملفات بايثون الأخرى، أو استخدامه في مترجم (interpreter) سطر أوامر بايثون.

يمكن للوحدات أن تعرِّف دوالًا وأصنافًا ومتغيرات يمكن الرجوع إليها من ملفات بايثون الأخرى، أو من مترجم سطر أوامر بايثون.

في بايثون، يمكنك الوصول إلى الوحدات باستخدام التعليمة `import`. عند فعل ذلك، ستُنفَّذ شيفرة الوحدة، مع الاحتفاظ بنطاقات (scopes) التعريفات حتى تكون متاحة في ملفك الحالي. فعندما تستورد بايثون وحدةً باسم `hello` على سبيل المثال، فسيبحث المترجم أولًا عن وحدة مضمَّنة باسم `hello`. فإن لم يجد، فسيبحث عن ملف يسمى `hello.py` في قائمة من المجلدات يحددها المتغير `sys.path`.

سيرشدك هذا الفصل إلى كيفية البحث عن الوحدات وتثبيتها، واستيرادها، وإعادة تسميتها (aliasing)، ثم سيعلمك كيفية إنشاء وحدة كاملة واستعمالها في شيفرة أخرى.

1. تثبيت الوحدات

هناك عدد من الوحدات المضمَّنة في مكتبة بايثون القياسية التي تحتوي على العديد من الوحدات التي توفر الكثير من وظائف النظام، أو توفر حلولًا قياسية. مكتبة بايثون القياسية تأتي مع كل توزيعات بايثون.

للتحقق من أنَّ وحدات بايثون جاهزة للعمل، ادخل إلى بيئة برمجة بايثون 3 المحلية، أو بيئة البرمجة المستندة إلى الخادم، وشغِّل مترجم بايثون في سطر الأوامر على النحو التالي:

```
(my_env) sammy@ubuntu:~/environment$ python
```

من داخل المترجم، يمكنك تنفيذ التعليمة `import` مع اسم الوحدة للتأكد من أنَّها جاهزة:

```
import math
```

لمَّا كانت `math` وحدة مضمَّنة، فينبغي أن يُكمل المترجم المهمة دون أي مشاكل، ثم يعود إلى المحث (prompt). هذا يعني أنَّك لست بحاجة إلى فعل أيِّ شيء للبدء في استخدام الوحدة `math`.

لننفِّذ الآن التعليمة `import` مع وحدة قد لا تكون مُثبَّتة عندك، مثل `matplotlib`، وهي مكتبة للرسم ثنائي الأبعاد:

```
import matplotlib
```

إذا لم تكن `matplotlib` مُثبَّتة، فسيظهر خطأٌ مثل الخطأ التالي:

```
ImportError: No module named 'matplotlib'
```

يمكنك إيقاف مترجم بايثون بالضغط على `CTRL + D`، ثم تثبيت الوحدة `matplotlib` عبر

pip بتنفيذ الأمر التالي:

```
(my_env) sammy@ubuntu:~/environment$ pip install matplotlib
```

بمجرد تثبيتها، يمكنك استيراد matplotlib من مترجم بايثون باستخدام import matplotlib، ولن يحدث أي خطأ.

2. استيراد الوحدات

للاستفادة من الدوال الموجودة في الوحدة، ستحتاج إلى استيراد الوحدة عبر التعليمة import، إذ تتألف من الكلمة المفتاحية import معقوبة باسم الوحدة كما في المثال الآتي.

يُصرَّح عن عملية استيراد الوحدات في أعلى ملفات بايثون، قبل الأسطر التوجيهية (shebang lines أي الأسطر التي تبدأ بـ !#)، أو التعليقات العامة. لذلك، سنستورد في ملف برنامج بايثون my_rand_int.py الوحدة random لتوليد أعداد عشوائية على النحو التالي:

```
import random
```

عندما نستورد وحدة، نجعلها بذلك متاحة في برنامجنا الحالي بوصفها فضاء اسم (namespace) منفصل. أي سيتعيَّن علينا الوصول إلى الدالة باستخدام الصياغة النقطية (dot notation) على النحو التالي [module].[function].

عمليًا، ستبدو الشيفرة باستخدام مثال الوحدة random كما يلي:

- random.randint(): تستدعي الدالة لإعادة عدد صحيح عشوائي، أو
- random.randrange(): تستدعي الدالة لإعادة عنصر عشوائي من نطاق محدد.

دعنا ننشئ حلقة for لتوضيح كيفية استدعاء دالة من الوحدة random ضمن

البرنامج my_rand_int.py:

```
import random

for i in range(10):
   print(random.randint(1, 25))
```

يستورد هذا البرنامج الصغير الوحدة random في السطر الأول، ثم ينتقل إلى الحلقة for التي ستمر على 10 عناصر. داخل الحلقة، سيطبع البرنامج عددًا صحيحًا عشوائيًا من المجال 1 إلى 25 (مشمول). يُمرِّر العددان الصحيحان 1 و 25 بوصفهما معاملات إلى random.randint().

عند تنفيذ البرنامج باستخدام الأمر python my_rand_int.py، ستظهر 10 أعداد صحيحة عشوائية في المخرجات. نظرًا لأنَّ هذه العناصر عشوائية، فستحصل على الأرجح على أعداد مختلفة (كلها محصورة بين 1 و 25) في كل مرة تنفِّذ فيها البرنامج، لكنَّها عمومًا ستبدو كما يلي:

```
6
9
1
14
3
22
10
1
15
9
```

إذا رغبت في اســتخدام دوال من أكـثر من وحـدة، يمكنـك ذلـك عن طريـق إضـافة عـدة تعليمات استيراد:

```
import random
import math
```

قد تصادف شيفرات تستورد عدة وحدات مفصولة بفواصل مثل `import random, math` ولكنَّ هذا لا يتوافق مع دليل التنسيق PEP 8.

للاستفادة من الوحـدة الإضـافيَّة، يمكننا إضـافة الثـابت `pi` من الوحـدة `math` إلى برنامجنا، وتقليل عدد الأعداد الصحيحة العشوائية المطبوعة:

```
import random
import math


for i in range(5):
    print(random.randint(1, 25))

print(math.pi)
```

الآن، عند تنفيـذ البرنامج، سنحصل على مخرجات على الشكل التالي، مع تقريب للعدد `pi` في السطر الأخير:

```
18
10
7
13
10
3.141592653589793
```

تتيح لك التعليمة `import` استيراد وحدة واحدة أو أكثر إلى برامجك، وهذا يمكّنك من الاستفادة مما تحويها تلك الوحدات.

3. استيراد عناصر محدَّدة

للإشارة إلى عناصر من وحدة مستوردة ضمن فضاء الأسماء، يمكنك استخدام التعليمة `from ... import`. عندما تستورد الوحدات بهذه الطريقة، سيكون بمقدورك الرجوع إلى الدوال بأسمائها فقط، بدلًا من استخدام الصياغة النقطية. في هذه الصياغة، يمكنك تحديد التعريفات التي تود الإشارة إليها مباشرة.

في بعض البرامج، قد ترى العبارة `from ... import *`، إذ تشير العلامة * إلى جميع العناصر الموجودة في الوحدة، ولكنّ هذه الصياغة غير معتمدة في PEP 8.

سنحاول في البداية استيراد دالة واحدة من الوحدة `random`، وهي `randint()`:

```
from random import randint
```

هنا، نستدعي أولًا الكلمة المفتاحية `from`، ثم `random`. بعد ذلك، نستخدم الكلمة المفتاحية `import`، ونستدعي الدالة المحددة التي نودُّ استخدامها.

الآن، عندما نرغب في استخدام هذه الدالة في برنامجنا، لن نستدعي الدالة وفق الصياغة النقطية، `random.randint()`، ولكن سنستدعيها باسمها مباشرةً، أي `randint()`:

```
from random import randint


for i in range(10):
    print(randint(1, 25))
```

عند تنفيذ البرنامج، ستتلقى مخرجات مشابهة لما تلقيته مسبقًا.

يتيح لنا استخدام `import ... from` الرجوع إلى العناصر المعرَّفة في الوحدة من فضاء الأسماء الخاص ببرنامجنا، مما يتيح لنا تجنب استخدام الصياغة النقطية الطويلة.

4. الأسماء المستعارة في الوحدات

يمكن إعطاء أسماء مستعارة للوحدات ودوالها داخل بايثون باستخدام الكلمة المفتاحية `as`.

قد ترغب في تغيير اسم ما لأنَّك تستخدمه سلفًا في برنامجك، أو أنَّه مستخدم في وحدة أخرى مستوردة، أو قد ترغب في اختصار اسم طويل تستخدمه كثيرًا. يمكنك ذلك عبر الصياغة التالية:

```
import [module] as [another_name]
```

لنعدِّل اسم الوحدة `math` في ملف البرنامج `my_math.py`. سنغيّر اسم الوحدة `math` إلى `m` من أجل اختصاره. سيبدو برنامجنا المعدل كالتالي:

```
import math as m


print(m.pi)
print(m.e)
```

سنشير داخل البرنامج إلى الثابت `pi` بالتعبير `m.pi`، بدلًا من `math.pi`.

يشيع في بعض الوحدات استخدام أسماء مستعارة (aliases) محدَّدة. فمثلًا، يدعو التوثيق الرسمي للوحدة `matplotlib.pyplot` إلى استخدام الاسم المستعار `plt`:

```
import matplotlib.pyplot as plt
```

يسمح هذا للمبرمجين بإلحاق الكلمة القصيرة `plt` بأي دالة متاحة داخل الوحدة، كما هو الحال في `plt.show()`.

5. كتابة وحدات مخصَّصة واستيرادها

سنتعلم الآن كتابة وحدات بايثون - بعد تعلم كيفية استيرادها - لاستخدامها في ملفات البرمجة الأخرى.

كتابة الوحدات مشابه لكتابة أي ملف بايثون آخر. يمكن أن تحتوي الوحدات على تعريفات الدوال والأصناف والمتغيِّرات التي يمكن استخدامها بعد ذلك في برامج بايثون الأخرى.

سننشئ من بيئة البرمجة الحالية الخاصة ببايثون 3 أو بيئة البرمجة المستندة إلى الخادم ملفًّا باسم `hello.py`، والذي سنستورده لاحقًا من ملف آخر.

في البدء، سننشئ دالة تطبع العبارة "!Hello, world":

```
# تعريف دالة
def world():
    print("Hello, world!")
```

إذا نفَّذنا البرنامج في سطر الأوامر باستخدام `python hello.py`، فلن يحدث شيء، لأنَّنا لم نطلب من البرنامج فعل أي شيء.

لننشئ ملفًا ثانيًا في نفس المجلد (أي بجانب الملف السابق) باسم `main_program.py` حتى نتمكن من استيراد الوحدة التي أنشأناها للتو، ومن ثم استدعاء الدالة. يجب أن يكون هذا الملف في نفس المجلد حتى تعرف بايثون موضع الوحدة، لأنَّها ليست وحدة مُضمَّنة ضمن مكتبة بايثون.

```
# استيراد الوحدة hello
import hello

# استدعاء الدالة
hello.world()
```

نظرًا لأننا استوردنا الوحدة، نحتاج إلى استدعاء الدالة من خلال الإشارة إلى اسم الوحدة بالصياغة النقطية (dot notation). يمكننا بدلًا من ذلك استيراد دالة محدَّدة من الوحدة بالتعليمة `from hello import world`، واستدعاء تلك الدالة بالشكل `world()` كما تعلمنا ذلك من الفصل السابق. الآن، يمكننا تنفيذ البرنامج من سطر الأوامر:

```
python main_program.py
```

سنحصل على المخرجات التالية:

```
Hello, world!
```

لنرى كيف يمكننا استخدام المتغيرات في الوحدات، دعنا نضيف تعريفًا لمتغيِّر في الملف `hello.py`:

```
# تعريف دالة
def world():
    print("Hello, world!")

# تعريف المتغير
shark = "Sammy"
```

بعد ذلك، سنستدعي المتغير داخل الدالة `print()` في الملف `main_program.py`:

```
# hello الوحدة استيراد
import hello

# الدالة استدعاء
hello.world()

# المتغير طباعة
print(hello.shark)
```

بمجرد تنفيذ البرنامج، سنحصل على المخرجات التالية:

```
Hello, world!
Sammy
```

أخيرًا، دعنا نعرِّف صنفًا في الملف `hello.py`. سننشئ الصنف `Octopus`، والذي يحتوي على الخاصيتين `name` و `color`، إضافة إلى دالة تطبع الخاصيات عند استدعائها:

```
# الدالة تعريف
def world():
   print("Hello, world!")

# المتغير تعريف
shark = "Sammy"
# الصنف تعريف
class Octopus:
   def __init__(self, name, color):
     self.color = color
     self.name = name

   def tell_me_about_the_octopus(self):
     print("This octopus is " + self.color + ".")
     print(self.name + " is the octopus's name.")
```

سنضيف الآن الصنفَ إلى نهاية الملف main_program.py:

```
# استيراد الوحدة hello
import hello


# استدعاء الدالة
hello.world()

# طباعة المتغير
print(hello.shark)

# استدعاء الصنف
jesse = hello.Octopus("Jesse", "orange")
jesse.tell_me_about_the_octopus()
```

بمجرد استدعاء الصنف Octopus باستخدام hello.Octopus()، يمكننا الوصول إلى دوال وخاصيات الصنف من فضاء الأسماء الخاص بالملف main_program.py. يتيح لنا هذا كتابة jesse.tell_me_about_the_octopus() في السطر الأخير دون استدعاء hello. يمكننا أيضًا، على سبيل المثال، استدعاء إحدى خاصيات الصنف، مثل jesse.color، دون الرجوع إلى اسم الوحدة hello.

سنحصل عند تنفيذ البرنامج التالي على المخرجات التالية:

```
Hello, world!
Sammy
This octopus is orange.
Jesse is the octopus's name.
```

من المهم أن تضع في الحسبان أن الوحدات لا تضم تعريفات دوال فقط، بل يمكن أيضًا أن

تقدم شيفرات برمجية وتنفِّذها. لتوضيح هذا، دعنا نعيد كتابة الملف hello.py لنجعله يقدم الدالة world() وينفذها أيضًا:

```
# تعريف دالة
def world():
    print("Hello, world!")

# استدعاء الدالة داخل الوحدة
world()
```

لقد حذفنا أيضًا التعريفات الأخرى في الملف. الآن، في الملف main_program.py، سنحذف كل الأسطر باستثناء عبارة الاستيراد:

```
# استيراد الوحدة hello
import hello
```

عند تنفيذ main_program.py، سنحصل على المخرجات التالية:

```
Hello, world!
```

هذا لأنَّ الوحدة hello قدمت الدالة world()، والتي مُرِّرت بعد ذلك إلى main_program.py لتُنفَّذ مع السكربت main_program.py.

الوحدة هي ملف بايثون مؤلف من تعريفات و شيفرات برمجية يمكن الاستفادة منها في ملفات بايثون الأخرى.

6. الوصول إلى الوحدات من مجلد آخر

قد تكون الوحدات مفيدة لأكثر من مشروع واحد، وفي هذه الحالة، لن يكون من الحكمة الاحتفاظ بالوحدة في مجلد مرتبط بمشروع خاص.

إذا أردت استخدام وحدة من مجلد آخر غير المجلد الذي يحوي البرنامج الرئيسي، فأمامك عدّة خيارات سنسردها فيما يلي.

ا. التعرف تلقائيًا على مسار الوحدة

أحد الخيارات هو استدعاء مسار الوحدة من الملفات البرمجية التي تستخدم تلك الوحدة. يُعد هذا حلًّا مؤقتًا يمكن استخدامه أثناء عملية التطوير، لأنَّه لا يجعل الوحدة متاحة على مستوى النظام بأكمله. لإلحاق مسار وحدة بملف برمجي آخر، ستبدأ باستيراد الوحدة sys، إلى جانب الوحدات الأخرى التي ترغب في استخدامها في ملف البرنامج الرئيسي.

تعد الوحدة sys جزءًا من مكتبة بايثون القياسية، وتوفر معاملات ودوال نظامية يمكنك استخدامها في برنامجك لتعيين مسار الوحدة التي ترغب في تقديمها. على سبيل المثال، لنقل أنَّنا نقلنا الملف hello.py إلى المسار /usr/sammy/، بينما يوجد الملف main_program.py في مجلد آخر. في الملف main_program.py، ما يزال بإمكاننا استيراد الوحدة hello عن طريق استيراد الوحدة sys، ثم إضافة المسار /usr/sammy/ إلى المسارات التي يبحث بايثون فيها عن الملفات.

```
import sys
sys.path.append('/usr/sammy/')
import hello
...
```

إن عيّنت مسار الملف hello.py كما يجب، فسيكون بمقدورك تنفيذ الملف main_program.py دون أيِّ أخطاء، وستحصل على نفس المخرجات التي حصلنا عليها أعلاه عندما كان الملف hello.py في نفس المجلد.

ب. إضافة الوحدة إلى مسار بايثون

الخيار الثاني هو إضافة الوحدة إلى المسار الذي يبحث فيه بايثون عن الوحدات والحزم، وهذا حل أفضل وأدوم، إذ يجعل الوحدة متاحةً على نطاق البيئة، أو على مستوى النظام.

لمعرفة المسار الذي يبحث فيه بايثون، شغِّل مترجم (interpreter) بايثون من بيئة البرمجة خاصتك:

```
python
```

بعد ذلك، استورد الوحدة sys:

```
import sys
```

ثم اطلب من بايثون طباعة مسار النظام:

```
print(sys.path)
```

ستحصل على بعض المخرجات، وسيُطبع مسار نظام واحد على الأقل. إذا كنت تعمل في بيئة برمجة، فقد تتلقى العديد منها. سيكون عليك البحث عن المسارات الموجودة في البيئة التي تستخدمها حاليًا، ولكن قد ترغب أيضًا في إضافة الوحدة إلى مسار النظام الرئيسي لبايثون. النتيجة ستكون مشابهة لما يلي:

```
'/usr/sammy/my_env/lib/python3.5/site-packages'
```

يمكنك الآن نقل الملف hello.py إلى هذا المجلد. بعد ذلك، يمكنك استيراد الوحدة hello مثل المعتاد:

```
import hello
...
```

عند تنفيذ البرنامج السابق، يُفترض ألا يحدث أيّ خطأ. يضمن لك تعديل مسار الوحدة إمكانية الوصول إليها مهما كان المجلد الذي تعمل فيه، إذ هذا مفيد خاصة في حال كنت تعمل على عدة مشاريع تشير إلى الوحدة نفسها.

7. خلاصة الفصل

يسمح مفهوم الوحدات باستدعاء دوال غير مضمَّنة في بايثون. فبعض الوحدات مُثبَّتة كجزء من بايثون، وبعضها سنثبّتها عبر pip، ويتيح لنا استخدام الوحدات توسيع برامجنا وتدعيمها، إذ تضع تحت تصرُّفنا شفرات جاهزة للاستخدام، فيمكننا إنشاء وحدات خاصة بنا، لنستخدمها نحن، أو المبرمجون الآخرون.

18

بناء الأصناف واستنساخ الكائنات

بايثون لغةٌ برمجة كائنية (object-oriented programming language). فتركّز البرمجة الكائنية (OOP) على كتابة شيفرات قابلة لإعادة الاستخدام، على عكس البرمجة الإجرائية (procedural programming) التي تركز على كتابة تعليمات صريحة ومتسلسلة.

تتيح البرمجة الكائنية لمبرمجي بايثون كتابة شيفرات سهلة القراءة والصيانة، وهذا مفيد للغاية عند تطوير البرامج المُعقَّدة.

التمييز بين الأصناف والكائنات أحدُ المفاهيم الأساسية في البرمجة الكائنية، ويوضح التعريفان التاليان الفرق بين المفهومين:

- الصنف (class): نموذج عام تُنسج على منواله كائنات يُنشِئها المبرمج. فيُعرِّف الصنف مجموعةً من الخاصيات التي تميز أي كائن يُستنسَخ (instantiated) منه.
- الكائن (object): نسخةٌ (instance) تُشتق من صنف، فهو تجسيد عملي للصنف داخل البرنامج.

تُستخدَم الأصناف لإنشاء أنماط، ثم تُستعمل تلك الأنماط لإنشاء الكائنات.

ستتعلم في هذا الفصل كيفيَّة إنشاء الأصناف والكائنات، وتهيئة الخاصيات باستخدام تابعٍ بانٍ (constructor method)، والعمل على أكثر من كائن من نفس الصنف.

1. الأصناف

الأصناف هي نماذج عامة تُستخدم لإنشاء كائنات، وسبق أن عرَّفناها آنفًا. تُنشَأ الأصناف باستخدام الكلمة المفتاحية `class`، بشكل مشابه لتعريف الدوال الذي يكون باستخدام الكلمة المفتاحية `def`.

دعنا نعرِّف صنفًا يسمى `Shark`، وننشئ له تابعين مرتبطين به، `swim` و `be_awesome`:

```
class Shark:
    def swim(self):
      print("The shark is swimming.")

    def be_awesome(self):
      print("The shark is being awesome.")
```

تُسمَّى مثل هذه الدوال «تابع» (method) لأنَّهما معرفتان داخل الصنف Shark؛ أي أنَّهما دالتان تابعتان للصنف Shark.

الوسيط الأول لهاتَين الدالتين هو self، وهو مرجع إلى الكائنات التي ستُبنَى من هذا الصنف. للإشارة إلى نُسخ (أو كائنات) من الصنف، يوضع self دائمًا في البداية، لكن يمكن أن تكون معه وسائط أخرى.

لا يؤدي تعريف الصنف Shark إلى إنشاء كائنات منه، وإنما يعرِّف فقط النمط العام لتلك الكائنات، والتي يمكننا تعريفها لاحقًا. لذا، إذا نفّذت البرنامج أعلاه الآن، فلن يُعاد أي شيء.

2. الكائنات

الكائن هو نسخةٌ (instance) من صنف، ويمكن أن نأخذ الصنف Shark المُعرَّف أعلاه، ونستخدمه لإنشاء كائن يعدُّ نسخةً منه.

سننشئ كائنًا من الصنف Shark باسم sammy:

```
sammy = Shark()
```

لقد أحلنا على الكائن sammy ناتج الباني Shark()، والذي يعيد نسخةً من الصنف.

سنستخدم في الشيفرة التالية التابعين الخاصين بالكائن sammy:

```
sammy = Shark()
sammy.swim()
sammy.be_awesome()
```

يستخدم الكائن sammy التابعين swim() و be_awesome()، وقد استدعيناهما باستعمال الصياغة النقطية (.)، والتي تُستخدَم للإشارة إلى خاصيات (properties) أو توابع (method) الكائنات. في هذه الحالة، استدعينا تابعًا، لذلك استعملنا قوسين مثلما نفعل عند استدعاء دالة.

الكلمة self هي معامل يُمرَّر إلى توابع الصنف Shark، في المثال أعلاه، يمثِّل self الكائن sammy. يتيح المعامل self للتوابع الوصول إلى خاصيات الكائن الذي استُدعيت معه.

لاحظ أنَّنا لم نمرِّر شيئًا داخل القوسين عند استدعاء التابع أعلاه، ذلك أنَّ الكائن sammy يُمرَّر تلقائيًا مع العامل النقطي. يوضِّح البرنامج التالي لنا الأمر:

```
class Shark:
    def swim(self):
      print("The shark is swimming.")

    def be_awesome(self):
      print("The shark is being awesome.")


def main():
    sammy = Shark()
    sammy.swim()
    sammy.be_awesome()

if __name__ == "__main__":
    main()
```

لننفِّذ البرنامج لنرى ما سيحدث:

```
python shark.py
```

ستُطبَع المخرجات التالية:

```
The shark is swimming.
The shark is being awesome.
```

في الشيفرة أعلاه، استدعى الكائن sammy التابعين swim() و be_awesome() في الدالة الرئيسية main().

3. الباني (Constructor)

يُستخدم الباني (Constructor Method) لتهيئة البيانات الأولية، ويُنفَّذ لحظة إنشاء الكائن.

في تعريف الصنف، يأخذ الباني الاسم __init__، وهو أول تابع يُعرَّف في الصنف، ويبدو كما يلي:

```
class Shark:
    def __init__(self):
        print("This is the constructor method.")
```

إذا أضفت التابع __init__ إلى الصنف Shark في البرنامج أعلاه، فسيَطبع البرنامجُ المخرجات التالية:

```
This is the constructor method.
The shark is swimming.
The shark is being awesome.
```

يُنفَّذ الباني تلقائيًا، لذا يستخدمه مطورو بايثون لتهيئة أصنافهم.

سنُعدِّل الباني أعلاه، ونجعله يستخدم متغيرًا اسمه name سيمثّل اسم الكائن. في الشيفرة التالية، سيكون المتغير name المعامل المُمرَّر إلى الباني، ونحيل قيمته إلى الخاصية self.name:

```
class Shark:
    def __init__(self, name):
        self.name = name
```

بعد ذلك، يمكننا تعديل السلاسل النصية في دوالنا للإشارة إلى اسم الصنف، على النحو التالي:

```
class Shark:
    def __init__(self, name):
        self.name = name

    def swim(self):
        # الإشارة إلى الاسم
        print(self.name + " is swimming.")

    def be_awesome(self):
        # الإشارة إلى الاسم
        print(self.name + " is being awesome.")
```

أخيرًا، يمكننا تعيين اسم الكائن sammy عند القيمة "Sammy" (أي قيمة الخاصية name) بتمريره إلى Shark() عند إنشائه:

```
class Shark:
    def __init__(self, name):
        self.name = name

    def swim(self):
        print(self.name + " is swimming.")
```

```
    def be_awesome(self):
        print(self.name + " is being awesome.")


def main():
    # تعيين اسم كائن Shark
    sammy = Shark("Sammy")
    sammy.swim()
    sammy.be_awesome()

if __name__ == "__main__":
    main()
```

عرّفنا التابع `__init__`، والذي يقبل مُعاملين `self` و `name` (تذكر أنّ المعامل `self` يُمرَّر تلقائيًا إلى التابع)، ثم عرَّفنا متغيرًا فيه.

عند تنفيذ البرنامج:

```
python shark.py
```

سنحصل على:

```
Sammy is swimming.
Sammy is being awesome.
```

لقد طُبِع الاسم الذي مرَّرناه إلى الكائن. ونظرًا لأنَّ الباني يُنفَّذ تلقائيًا، فلست بحاجة إلى استدعائه بشكل صريح، فيكفي تمرير الوسائط بين القوسين التاليين لاسم الصنف عند إنشاء نسخة جديدة منه.

إذا أردت إضافة معامل آخر، مثل `age`، فيمكن ذلك عبر تمريره إلى التابع `__init__`:

```
class Shark:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

عند إنشاء الكائن `sammy`، سنمرِّر عُمره أيضًا بالإضافة إلى اسمه:

```
sammy = Shark("Sammy", 5)
```

إذًا، تتيح البانيات تهيئة خاصيات الكائن لحظة إنشائه.

4. العمل مع عدة كائنات

تتيح لنا الأصناف إنشاء العديد من الكائنات المتماثلة التي تتبع نفس النمط. لتفهم ذلك بشكل أفضل، دعنا نضيف كائنًا آخر من الصنف `Shark` إلى برنامجنا:

```
class Shark:
    def __init__(self, name):
        self.name = name

    def swim(self):
        print(self.name + " is swimming.")

    def be_awesome(self):
        print(self.name + " is being awesome.")

def main():
    sammy = Shark("Sammy")
    sammy.be_awesome()
    stevie = Shark("Stevie")
    stevie.swim()
```

```
if __name__ == "__main__":
  main()
```

لقد أنشأنا كائنًا ثانيًا من الصنف Shark يسمى stevie ومرّرنا إليه الاسم "Stevie".

استدعينا في هذا المثال التابع be_awesome()‎ مع الكائن sammy والتابع swim()‎ مع الكائن stevie. سننفذ البرنامج عبر الأمر التالي:

```
python shark.py
```

سنحصل على المخرجات التالية:

```
Sammy is being awesome.
Stevie is swimming.
```

يبدو ظاهرًا في المخرجات أننا نستخدم كائنين مختلفين، الكائن sammy والكائن stevie، وكلاهما من الصنف Shark.

تتيح لنا الأصناف إنشاء عدة كائنات تتبع كلها نفس النمط دون الحاجة إلى بناء كل واحد منها من البداية.

5. فهم متغيرات الأصناف والنسخ

تسمح البرمجة الكائنية باستخدام متغيرات على مستوى الصنف، أو على مستوى النسخة (instance). المتغيرات هي رموز (symbols) تدل على قيمة تستخدمها في برنامجك.

يشار إلى المتغيرات على مستوى الصنف باسم «متغيرات الصنف» (class variables)، في حين تسمى المتغيرات الموجودة على مستوى النسخة باسم «متغيرات النسخة» (instance variables).

إذا توقعت أن يكون المتغيّر متسقًا في جميع نسخ الصنف، أو عندما تود تهيئة المتغير، فالأفضل أن تُعرِّف ذلك المتغير على مستوى الصنف. أمّا إن كنت تعلم أنَّ المتغير سيختلف من نسخة إلى أخرى، فالأفضل أن تُعرِّفَه على مستوى النسخة. يسعى أحد مبادئ تطوير البرمجيات هو مبدأ DRY (اختصارًا للعبارة don’t repeat yourself، والذي يعني لا تكرِّر نفسك) إلى الحد من التكرار في الشيفرة. أي تلتزم البرمجة الكائنية بمبدأ DRY على تقليل التكرار في الشيفرة.

ا. متغيرات الصنف

تُعرَّف متغيرات الصنف داخل الصنف وخارج كل توابعه وعادةً ما توضع مباشرة أسفل ترويسة الصنف، وقبل الباني (constructor) والتوابع الأخرى. ولمَّا كانت مملوكة للصنف نفسه، فستُشارَك مع جميع نُسَخ ذلك الصنف. وبالتالي، سيكون لها نفس القيمة بغض النظر عن النسخة، إلا إن كنت ستستخدم متغير الصنف لتهيئة متغير معيَّن.

متغير الصنف يبدو كما يلي:

```
class Shark:
    animal_type = "fish"
```

في الشيفرة أعلاه أحلنا القيمة "fish" إلى المتغير animal_type.

يمكننا إنشاء نسخة من الصنف Shark (سنطلق عليها new_shark)، ونطبع المتغير باستخدام الصياغة النقطية (dot notation):

```
class Shark:
    animal_type = "fish"

new_shark = Shark()
print(new_shark.animal_type)
```

لننفذ البرنامج:

```
python shark.py
```

سيعيد البرنامج قيمة المتغير:

```
fish
```

دعنا نضيف مزيدًا من متغيرات الصنف، ونطبعها:

```
class Shark:
    animal_type = "fish"
    location = "ocean"
    followers = 5

new_shark = Shark()
print(new_shark.animal_type)
print(new_shark.location)
print(new_shark.followers)
```

يمكن أن تتألف متغيرات الصنف من أي نوع من البيانات المتاحة في بايثون تمامًا مثل أي متغير آخر. استخدمنا في هذا البرنامج السلاسل النصية والأعداد الصحيحة. لننفذ البرنامج مرة أخرى باستخدام الأمر `python shark.py` ونرى المخرجات:

```
fish
ocean
5
```

يمكن للنسخة `new_shark` الوصول إلى جميع متغيرات الصنف وطباعتها عند تنفيذ البرنامج، إذ تُنشأ عند إنشاء الصنف مباشرةً (وليس عند إنشاء نسخة منه) وتحتل موضعًا لها في الذاكرة ويمكن لأي كائن مُشتَق (نسخة) من الصنف نفسه أن يصل إليها ويقرأ قيمتها.

ب. متغيرات النسخة

تختلف متغيرات النسخة عن متغيرات الصنف بأن النسخة المشتقة من الصنف هي من تملكها وليس الصنف نفسه أي تكون على مستوى النسخة وسيُنشَأ متغيِّرٌ مستقل في الذاكرة عند إنشاء كل نسخة. هذا يعني أنَّ متغيرات النسخة ستختلف من كائن إلى آخر.

تُعرَّف متغيرات النسخة ضمن التوابع على خلاف متغيرات الصنف. في مثال الصنف Shark أدناه، عرفنا متغيري النسخة name و age:

```
class Shark:
   def __init__(self, name, age):
     self.name = name
     self.age = age
```

عندما ننشئ كائنًا من الصنف Shark، سيتعيَّن علينا تعريف هذه المتغيرات، عبر تمريرها معاملاتٍ ضمن الباني (constructor)، أو أي تابع آخر:

```
class Shark:
   def __init__(self, name, age):
     self.name = name
     self.age = age

new_shark = Shark("Sammy", 5)
```

كما هو الحال مع متغيرات الأصناف، يمكننا بالمثل طباعة متغيرات النسخة:

```
class Shark:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
new_shark = Shark("Sammy", 5)
print(new_shark.name)
print(new_shark.age)
```

عند تنفيذ البرنامج أعلاه باستخدام python shark.py، سنحصل على المخرجات التالية:

```
Sammy
5
```

تتألف المخرجات التي حصلنا عليها من قيم المتغيرات التي هيَّأناها لأجل الكائن new_shark.

لننشئ كائنًا آخر من الصنف Shark يسمى stevie:

```
class Shark:
    def __init__(self, name, age):
        self.name = name
        self.age = age

new_shark = Shark("Sammy", 5)
print(new_shark.name)
print(new_shark.age)

stevie = Shark("Stevie", 8)
print(stevie.name)
print(stevie.age)
```

يمرِّر الكائن stevie المعاملات إلى الباني لتعيين قيم متغيرات النسخة الخاصة به.

تسمح متغيرات النسخة، المملوكة لكائنات الصنف، لكل كائن أو نسخة أن تكون لها متغيرات خاصة بها ذات قيم مختلفة عن بعضها بعضًا.

6. العمل مع متغيرات الصنف والنسخة معًا

غالبًا ما تُستخدم متغيرات الصنف ومتغيرات النسخة في نفس الشيفرة، ويوضح المثال التالي يستخدم الصنف Shark الذي أنشأناه سابقًا هذا الأمر. تشرح التعليقات في البرنامج كل خطوة من خطوات العملية:

```
class Shark:

    # متغيرات الصنف
    animal_type = "fish"
    location = "ocean"

    # باني مع متغيري النسخة age و name
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # تابع مع متغير النسخة followers
    def set_followers(self, followers):
        print("This user has " + str(followers) + " followers")

def main():
    # الكائن الأول، إعداد متغيرات النسخة في الباني
    sammy = Shark("Sammy", 5)

    # طباعة متغير النسخة name
    print(sammy.name)

    # طباعة متغير الصنف location
    print(sammy.location)

    # الكائن الثاني
```

```
    stevie = Shark("Stevie", 8)

    # name طباعة متغير النسخة
    print(stevie.name)

    # set_followers استخدام التابع
    # followers لتمرير متغير النسخة
    stevie.set_followers(77)

    # animal_type طباعة متغير الصنف
    print(stevie.animal_type)

if __name__ == "__main__":
    main()
```

عند تنفيذ البرنامج باستخدام `python shark.py`، سنحصل على المخرجات التالية:

```
Sammy
ocean
Stevie
This user has 77 followers
fish
```

7. خلاصة الفصل

تطرَّقنا في هذا الفصل إلى عِدَّة مفاهيم، مثل إنشاء الأصناف، وإنشاء الكائنات، وتهيئة الخاصيات باستخدام البانيات، والعمل مع أكثر من كائن من نفس الصنف.

تُعدُّ البرمجة الكائنية أحد المفاهيم الضرورية التي ينبغي أن يتعلمها كل مبرمجي بايثون، إذ تساعد على كتابة شيفرات قابلة لإعادة الاستخدام، والكائنات التي تُنشَأ في برنامج ما يمكن استخدامها في برامج أخرى. كما أنَّ البرامج الكائنية عادة ما تكون أوضح وأكثر مقروئية،

خصوصًا في البرامج المعقدة التي تتطلب تخطيطا دقيقا، وهذا بدوره يسهل صيانة البرامج مستقبلا.

في البرمجة الكائنية، يشار إلى المتغيرات المُعرَّفة على مستوى الصنف بمتغيرات الصنف، في حين تسمى المتغيرات المُعرّفة على مستوى الكائن بمتغيرات النسخة. يتيح لنا هذا التمييز استخدام متغيرات ذات قيم واحدة بينها عبر متغيرات الصنف، أو استخدام متغيرات مختلفة لكل كائن على حدة عبر متغيرات النسخة. كما يضمن استخدام المتغيرات الخاصة بالصنف أو النسخة أن تكون الشيفرة متوافقة مع مبدأ DRY.

19

مفهوم الوراثة في البرمجة

تُسِّهل البرمجة الكائنية كتابة شيفرات قابلة لإعادة الاستخدام وتجنب التكرار في مشاريع التطوير. إحدى الآليات التي تحقِّق بها البرمجة الكائنية هذا الهدف هي مفهوم الوراثة (inheritance)، التي بفضلها يمكن لصنفٍ فرعي (subclass) استخدام الشيفرة الخاصة بصنف أساسي (base class، ويطلق عليه «صنف أب» أيضًا) موجود مسبقًا.

سيستعرض هذا الفصل بعض الجوانب الرئيسية لمفهوم الوراثة في بايثون، بما في ذلك كيفيَّة إنشاء الأصناف الأساسية (parent classes) والأصناف الفرعية (child classes)، وكيفيَّة إعادة تعريف (override) التوابع والخاصيات، وكيفيَّة استخدام التابع `super()`، وكيفيَّة الاستفادة من الوراثة المُتعدِّدة (multiple inheritance).

1. ما هي الوراثة؟

تقوم الوراثة على استخدام شيفرة صنف معين في صنف آخر أي يرث صنف يراد إنشاؤه شيفرة صنف آخر. يمكن تمثيل مفهوم الوراثة في البرمجة بالوراثة في علم الأحياء تمامًا، فالأبناء يرثون خاصيات معينة من آبائهم. ويمكن لطفل أن يرث طول والده أو لون عينيه بالإضافة إلى خاصيات أخرى جديدة خاصة فيه. كما يتشارك الأطفال نفس اسم العائلة الخاصة بآبائهم.

ترث الأصناف الفرعية (subclasses، تُسمى أيضًا الأصناف الأبناء [child classes]) التوابع والمتغيرات من الأصناف الأساسية (base classes، تسمى أيضًا الأصناف الآباء [parent classes]).

مثلًا، قد يكون لدينا صنف أساسي يسمى `Parent` يحوي متغيرات الأصناف `last_name` و `height` و `eye_color`، والتي سيرثها الصنف الابن `Child`.

لمّا كان الصنف الفرعي `Child` يرث الصنف الأساسي `Parent`، فبإمكانه إعادة استخدام شيفرة `Parent`، مما يسمح للمبرمج بكتابة شيفرة أوجز، وتقليل التكرار.

2. الأصناف الأساسية

تشكل الأصناف الأساسية أساسًا يمكن أن تستند إليه الأصناف الفرعية المُتفرّعة منها، إذ تسمح الأصناف الأساسية بإنشاء أصناف فرعية عبر الوراثة دون الحاجة إلى كتابة نفس الشيفرة في كل مرة. يمكن تحويل أي صنف إلى صنف أساسي، إذ يمكن استخدامه لوحده، أو جعله قالبًا (نموذجًا).

لنفترض أنَّ لدينا صنفًا أساسيًا باسم `Bank_account`، وصنفين فرعيين مُشتقين منه باسم `Personal_account` و `Business_account`. ستكون العديد من التوابع مشتركة بين الحسابات الشخصية (Personal_account) والحسابات التجارية (Business_account)، مثل توابع سحب وإيداع الأموال، لذا يمكن أن تنتمي تلك التوابع إلى الصنف الأساسي `Bank_account`. سيكون للصنف `Business_account` توابع خاصة به، مثل تابع مخصص لعملية جمع سجلات ونماذج الأعمال، بالإضافة إلى متغير `employee_identification_number` موروث من الصنف الأب.

وبالمثل، قد يحتوي الصنف `Animal` على التابعين `eating()` و `sleeping()`، وقد يتضمن الصنف الفرعي `Snake` تابعين إضافيين باسم `hissing()` و `slithering()` خاصين به.

دعنا ننشئ صنفًا أساسيًا باسم `Fish` لاستخدامه لاحقًا أساسًا لأصناف فرعية تمثل أنواع الأسماك. سيكون لكل واحدة من تلك الأسماك أسماء أولى وأخيرة، بالإضافة إلى خصائص مميزة خاصة بها.

سننشئ ملفًا جديدًا يسمى fish.py ونبدأ بالباني، والذي سنعرّف داخله متغيري الصنف first_name و last_name لكل كائنات الصنف Fish، أو أصنافه الفرعية.

```
class Fish:
    def __init__(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name
```

القيمة الافتراضية للمتغير last_name هي السلسلة النصية "Fish"، لأننا نعلم أنّ معظم الأسماك سيكون هذا هو اسمها الأخير.

لنُضِف بعض التوابع الأخرى:

```
class Fish:
    def __init__(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

لقد أضفنا التابعين swim() و swim_backwards() إلى الصنف Fish حتى يتسنى لكل الأصناف الفرعية استخدام هذه التوابع.

ما دام أنّ معظم الأسماك التي ننوي إنشاءها ستكون عظمية (أي أنّ لها هيكلا عظميًا) وليس غضروفية (أي أن لها هيكلًا غضروفيًا)، فيمكننا إضافة بعض الخاصيات الإضافية إلى التابع __init__():

```
class Fish:
   def __init__(self, first_name, last_name="Fish",
         skeleton="bone", eyelids=False):
     self.first_name = first_name
     self.last_name = last_name
     self.skeleton = skeleton
     self.eyelids = eyelids

   def swim(self):
     print("The fish is swimming.")

   def swim_backwards(self):
     print("The fish can swim backwards.")
```

لا يختلف بناء الأصناف الأساسية عن بناء أي صنف آخر، إلا أنَّنا نصممها لتستفيد منها الأصناف الفرعية المُعرّفة لاحقًا.

3. الأصناف الفرعية

الأصناف الفرعية هي أصناف ترث كل شيء من الصنف الأساسي. هذا يعني أنَّ الأصناف الفرعية قادرة على الاستفادة من توابع ومتغيرات الصنف الأساسي.

على سبيل المثال، سيتمكن الصنف الفرعي Goldfish المشتق من الصنف Fish من استخدام التابع swim() المُعرّف في Fish دون الحاجة إلى التصريح عنه.

يمكننا النظر إلى الأصناف الفرعية على أنَّها أقسامٌ من الصنف الأساسي. فإذا كان لدينا صنفًا فرعيًّا يسمى Rhombus (معيَّن)، وصنفًا أساسيًّا يسمى Parallelogram (متوازي الأضلاع)، يمكننا القول أنَّ المعين (Rhombus) هو متوازي أضلاع (Parallelogram).

يبدو السطر الأول من الصنف الفرعي مختلفًا قليلًا عن الأصناف غير الفرعية، إذ يجب عليك تمرير الصنف الأساسي إلى الصنف الفرعي كمعامل:

```
class Trout(Fish):
```

الصنف Trout هو صنف فرعي من Fish. يدلنا على هذا الكلمةُ Fish المُدرجة بين قوسين.

يمكننا إضافة توابع جديدة إلى الأصناف الفرعية، أو إعادة تعريف التوابع الخاصة بالصنف الأساسي، أو يمكننا ببساطة قبول التوابع الأساسية الافتراضية باستخدام الكلمة المفتاحية pass، وهو ما سنفعله في المثال التالي:

```
...
class Trout(Fish):
    pass
```

يمكننا الآن إنشاء كائن من الصنف Trout دون الحاجة إلى تعريف أي توابع إضافية.

```
...
class Trout(Fish):
    pass

terry = Trout("Terry")
print(terry.first_name + " " + terry.last_name)
print(terry.skeleton)
print(terry.eyelids)
terry.swim()
terry.swim_backwards()
```

لقد أنشأنا كائنًا باسم terry من الصنف Trout، والذي سيستخدم جميع توابع الصنف Fish وإن لم نعرِّفها في الصنف الفرعي Trout. يكفي أن نمرِّر القيمة "Terry" إلى المتغير first_name، أمَّا المتغيرات الأخرى فقد جرى تهيئتها سلفًا.

عند تنفيذ البرنامج، سنحصل على المخرجات التالية:

```
Terry Fish
bone
False
The fish is swimming.
The fish can swim backwards.
```

لننشئ الآن صنفًا فرعيًا آخر يعرّف تابعًا خاصا به. سنسمي هذا الصنف Clownfish. سيسمح التابع الخاص به بالتعايش مع شقائق النعمان البحري:

```
...
class Clownfish(Fish):

    def live_with_anemone(self):
      print("The clownfish is coexisting with sea anemone.")
```

دعنا ننشئ الآن كائنًا آخر من الصنف Clownfish:

```
...
casey = Clownfish("Casey")
print(casey.first_name + " " + casey.last_name)
casey.swim()
casey.live_with_anemone()
```

عند تنفيذ البرنامج، سنحصل على المخرجات التالية:

```
Casey Fish
The fish is swimming.
The clownfish is coexisting with sea anemone.
```

تُظهر المخرجات أنَّ الكائن casey المستنسخ من الصنف Clownfish قادر على استخدام التابعين ‎__init__()‎ و ‎swim()‎ الخاصين بالصنف Fish، إضافة إلى التابع

live_with_anemone() الخاص بالصنف الفرعي.

إذا حاولنا استخدام التابع live_with_anemone() في الكائن Trout، فسوف يُطلق خطأ:

```
terry.live_with_anemone()
AttributeError: 'Trout' object has no attribute
'live_with_anemone'
```

ذلك أنَّ التابع live_with_anemone() ينتمي إلى الصنف الفرعي Clownfish فقط، وليس إلى الصنف الأساسي Fish.

ترث الأصناف الفرعية توابع الصنف الأساسي الذي اشتُقَّت منه، لذا يمكن لكل الأصناف الفرعية استخدام تلك التوابع.

4. إعادة تعريف توابع الصنف الأساسي

عرَّفنا في المثال السابق الصنف الفرعي Trout الذي استخدم الكلمة المفتاحية pass ليرث جميع سلوكيات الصنف الأساسي Fish، وعرّفنا كذلك صنفًا آخر Clownfish يرث جميع سلوكيات الصنف الأساسي، ويُنشئ أيضًا تابعًا خاصًا به. قد نرغب في بعض الأحيان في استخدام بعض سلوكيات الصنف الأساسي، ولكن ليس كلها. يُطلَق على عملية تغيير توابع الصنف الأساسي «إعادة التعريف» (Overriding).

عند إنشاء الأصناف الأساسية أو الفرعية، فلا بد أن تكون لك رؤية عامة لتصميم البرنامج حتى لا تعيد تعريف التوابع إلا عند الضرورة.

سننشئ صنفًا فرعيًا Shark مشتقًا من الصنف الأساسي Fish، الذي سيمثل الأسماك العظمية بشكل أساسي، لذا يتعين علينا إجراء تعديلات على الصنف Shark المخصَّص في الأصل للأسماك الغضروفية. من منظور تصميم البرامج، إذا كانت لدينا أكثر من سمكة غير عظمية

واحدة، فيُستحب أن ننشئ صنفًا خاصًا بكل نوع من هذين النوعين من الأسماك.

تمتلك أسماك القرش، على عكس الأسماك العظمية، هياكل مصنوعة من الغضاريف بدلًا من العظام. كما أنَّ لديها جفونًا، ولا تستطيع السباحة إلى الوراء، كما أنَّها قادرة على المناورة للخلف عن طريق الغوص.

على ضوء هذه المعلومات، سنعيد تعريف الباني `__init__()` والتابع `swim_backwards()`. لا نحتاج إلى تعديل التابع `swim()` لأنَّ أسماك القرش يمكنها السباحة. دعنا نلقي نظرة على هذا الصنف الفرعي:

```
...
class Shark(Fish):
    def __init__(self, first_name, last_name="Shark",
                 skeleton="cartilage", eyelids=True):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids

    def swim_backwards(self):
        print("The shark cannot swim backwards, but can sink
backwards.")
```

لقد أعدنا تعريف المعاملات التي تمت تهيئتها في التابع `__init__()`، فأخذ المتغير `last_name` القيمة `"Shark"`، كما أُسنِد إلى المتغير `skeleton` القيمة `"cartilage"`، فيما أُسنِدَت القيمة المنطقية `True` إلى المتغير `eyelids`. يمكن لجميع نُسخ الصنف إعادة تعريف هذه المعاملات.

يطبع التابع `swim_backwards()` سلسلة نصية مختلفة عن تلك التي يطبعها في الصنف الأساسي `Fish`، لأنَّ أسماك القرش غير قادرة على السباحة للخلف كما تفعل الأسماك العظمية.

يمكننا الآن إنشاء نسخة من الصنف الفرعي `Shark`، والذي سيستخدم التابع `swim()` الخاص بالصنف الأساسي `Fish`:

```
...
sammy = Shark("Sammy")
print(sammy.first_name + " " + sammy.last_name)
sammy.swim()
sammy.swim_backwards()
print(sammy.eyelids)
print(sammy.skeleton)
```

عند تنفيذ هذه الشيفرة، سنحصل على المخرجات التالية:

```
Sammy Shark
The fish is swimming.
The shark cannot swim backwards, but can sink backwards.
True
cartilage
```

لقد أعاد الصنف الفرعي `Shark` تعريف التابعين `__init__()` و `swim_backwards()` الخاصين بالصنف الأساسي `Fish`، وورث في نفس الوقت التابع `swim()` الخاص بالصنف الأساسي.

5. الدالة super() وفائدتها في الوراثة

يمكنك باستخدام الدالة `super()` الوصول إلى التوابع الموروثة التي أُعيدَت كتابتها. عندما نستخدم الدالة `super()`، فإنَّنا نستدعي التابع الخاص بالصنف الأساسي لاستخدامه في الصنف

الفرعي. على سبيل المثال، قد نرغب في إعادة تعريف جانب من التابع الأساسي وإضافة وظائف معينة إليه، ثم بعد ذلك نستدعي التابع الأساسي لإنهاء بقية العمل.

في برنامج خاص بتقييم الطلاب مثلًا، قد نرغب في تعريف صنف فرعي `Weighted_grade` يرث الصنف الأساسي `Grade`، ونعيد فيه تعريف التابع `calculate_grade()` الخاص بالصنف الأساسي من أجل تضمين شيفرة خاصة بحساب التقدير المرجَّح (weighted grade)، مع الحفاظ على بقية وظائف الصنف الأساسي. عبر استدعاء التابع `super()`، سنكون قادرين على تحقيق ذلك.

عادة ما يُستخدم التابع `super()` ضمن التابع `__init__()`، لأنَّه المكان الذي ستحتاج فيه على الأرجح إلى إضافة بعض الوظائف الخاصة إلى الصنف الفرعي قبل إكمال التهيئة من الصنف الأساسي.

لنضرب مثلًا لتوضيح ذلك، دعنا نعدِّل الصنف الفرعي `Trout`. نظرًا لأنَّ سمك السلمون المرقَّط من أسماك المياه العذبة، فلنضف متغيرًا اسمه `water` إلى التابع `__init__()`، ولنُعطه القيمة `"freshwater"`، ولكن مع الحفاظ على باقي متغيرات ومعاملات الصنف الأساسي:

```
...
class Trout(Fish):
    def __init__(self, water = "freshwater"):
        self.water = water
        super().__init__(self)
...
```

لقد أعدنا تعريف التابع `__init__()` في الصنف الفرعي `Trout`، وغيرنا سلوكه موازنةً بالتابع `__init__()` المُعرَّف سلفًا في الصنف الأساسي `Fish`. لاحظ أنَّنا استدعينا

التابع ()__init__ الخاص بالصنف Fish صراحةً ضمن التابع ()__init__ الخاص بالصنف Trout.

بعد إعادة تعريف التابع، لم نعد بحاجة إلى تمرير first_name معاملًا إلى Trout، وفي حال فعلنا ذلك، فسيؤدي ذلك إلى إعادة تعيين freshwater بدلاً من ذلك. ستُهيِّئ بعد ذلك الخاصية first_name عن طريق استدعاء المتغير في الكائن خاصتنا.

الآن يمكننا استدعاء متغيرات الصنف الأساسي التي أُعدِّت، وكذلك استخدام المتغيِّر الخاص بالصنف الفرعي:

```
...
terry = Trout()

# تهيئة الاسم الأول
terry.first_name = "Terry"

# super() استخدام __init__() الخاص بالصنف الأساسي عبر
print(terry.first_name + " " + terry.last_name)
print(terry.eyelids)

# استخدام __init__() المعاد تعريفها في الصنف الفرعي
print(terry.water)

# استخدام التابع swim() الخاص بالصنف الأساسي
terry.swim()
```

سنحصل على المخرجات التالية:

```
Terry Fish
False
freshwater
```

```
The fish is swimming.
```

تُظهر المخرجات أنَّ الكائن terry المنسوخ من الصنف الفرعي Trout قادر على استخدام المتغير water الخاص بتابع الصنف الفرعي __init__()، إضافة إلى استدعاء المتغيرات first_name و last_name و eyelids الخاصة بالتابع __init__() المُعرَّف في الصنف الأساسي Fish.

يسمح لنا التابع super() المُضمن في بايثون باستخدام توابع الصنف الأساسي حتى بعد إعادة تعريف تلك التوابع في الأصناف الفرعية.

6. الوراثة المُتعدِّدة (Multiple Inheritance)

المقصود بالوراثة المتعددة هي قدرة الصنف على أن يرث الخاصيات والتوابع من أكثر من صنف أساسي واحد. هذا من شأنه تقليل التكرار في البرامج، ولكنَّه قد يُعقِّد العمل، لذلك يجب استخدام هذا المفهوم بحذر.

لإظهار كيفية عمل الوراثة المتعدِّدة، دعنا ننشئ صنفًا فرعيًا Coral_reef يرث من الصنفين Coral و Sea_anemone. يمكننا إنشاء تابع في كل صنف أساسي، ثم استخدام الكلمة المفتاحية pass في الصنف الفرعي Coral_reef:

```
class Coral:

    def community(self):
        print("Coral lives in a community.")

class Anemone:

    def protect_clownfish(self):
```

```
        print("The anemone is protecting the clownfish.")


class CoralReef(Coral, Anemone):
    pass
```

يحتوي الصنف Coral على تابع يسمى community()، والذي يطبع سطرًا واحدًا، بينما يحتوي الصنف Anemone على تابع يسمى protect_clownfish()، والذي يطبع سطرًا آخر. سنُمرِّر الصنفين كلاهما بين قوسين في تعريف الصنف CoralReef، ما يعني أنَّه سيرث الصنفين معًا.

دعنا الآن ننشئ كائنًا من الصنف CoralReef:

```
...
great_barrier = CoralReef()
great_barrier.community()
great_barrier.protect_clownfish()
```

الكائن great_barrier مُشتقٌّ الصنف CoralReef، ويمكنه استخدام التوابع من كلا الصنفين الأساسيين. عند تنفيذ البرنامج، سنحصل على المخرجات التالية:

```
Coral lives in a community.
The anemone is protecting the clownfish.
```

تُظهِر المخرجات أنَّ التوابع من كلا الصنفين الأساسيين استُخدِما بفعالية في الصنف الفرعي.

تسمح لنا الوراثة المُتعدِّدة بإعادة استخدام الشيفرات البرمجية المكتوبة في أكثر من صنف أساسي واحد. وإذا تم تعريف التابع نفسه في أكثر من صنف أساسي واحد، فسيستخدم الصنف

الفرعي التابع الخاص بالصنف الأساسي الذي ظهر أولًا في قائمة الأصناف المُمرَّرة إليه عند تعريفه.

رغم فوائدها الكثيرة وفعاليتها، إلا أنَّ عليك توخي الحذر في استخدام الوراثة المُتعدِّدة، حتى لا ينتهي بك الأمر بكتابة برامج مُعقَّدة وغير مفهومة للمبرمجين الآخرين.

7. خلاصة الفصل

تعلمنا في هذا الفصل كيفيَّة إنشاء أصناف أساسية وفرعيَّة، وكيفيَّة إعادة تعريف توابع وخاصيات الأصناف الأساسية داخل الأصناف الفرعية باستخدام التابع `super()`، إضافة إلى مفهوم الوراثة المتعددة.

الوراثة هي إحدى أهم ميزات البرمجة الكائنية التي تجعلها متوافقة مع مبدأ DRY (لا تكرر نفسك)، وهذا يحسن إنتاجية المبرمجين، ويساعدهم على تصميم برامج فعالة وواضحة.

20

التعددية الشكلية وتطبيقاتها

التعددية الشكلية (Polymorphism) هي القدرة على استخدام واجهة موحدة لعدة أشكال مختلفة، مثل أنواع البيانات أو الأصناف، وهذا يسمح للدوال باستخدام كيانات من أنواع مختلفة. بالنسبة للبرامج الكائنية في بايثون، هذا يعني أنَّه يمكن استخدام كائن معين ينتمي إلى صنف مُعيَّن كما لو كان ينتمي إلى صنف مختلف. تسمح التعددية الشكلية بكتابة شيفرات مرنة ومجرَّدَة وسهلة التوسيع والصيانة.

سوف تتعلم في هذا الفصل كيفية تطبيق التعددية الشكلية على أصناف بايثون.

1. ما هي التعددية الشكلية (Polymorphism)؟

التعددية الشكلية هي إحدى السمات الأساسية للأصناف في بايثون، وتُستخدَم عندما تكون هناك توابع لها نفس الأسماء في عدة أصناف، أو أصناف فرعية. يسمح ذلك للدوال باستخدام كائنات من أيٍّ من تلك الأصناف والعمل عليها دون الاكتراث لنوعها.

يمكن تنفيذ التعددية الشكلية عبر الوراثة، أو باستخدام توابعِ الأصناف الفرعية، أو إعادة تعريفها (overriding).

يستخدم بايثون نظام أنواع (typing) خاص، يسمى «نظام التحقق من الأنواع: البطة نموذجًا» (Duck Typing)، وهو حالة خاصة من أنظمة التحقق من الأنواع الديناميكية (Dynamic Typing). يستخدم هذا النظامُ التعدُّديةَ الشكلية، بما في ذلك الربط المتأخر (late binding)، والإيفاد الديناميكي (Dynamic dispatch). يعتمد هذا النظام على «نموذج البطة» بناءً على اقتباسٍ للكاتب جيمس ويتكومب رايلي:

> «عندما أرى طائرًا يمشي مثل بطة، ويسبح مثل بطة، وصوته كصوت البطة، فسأعدُّ هذا الطائر بطةً»

خُصِّص هذا المفهوم من قبل مهندس الحاسوب الإيطالي أليكس مارتيلي (Alex Martelli) في رسالة إلى مجموعة comp.lang.python، يقوم نظام التحقق من الأنواع هذا الذي يعتمد البطة نموذجًا على تعريف الكائن من منظور ملاءمة الغرض الذي أُنشِئ لأجله. عند استخدام نظام أنواع عادي، فإنَّ ملاءمة الكائن لغرض مُعيَّن يتحدد بنوع الكائن فقط، ولكن في نموذج البطة، يَتحدَّد ذلك بوجود التوابع والخاصيات الضرورية لذلك الغرض بدلًا من النوع الحقيقي للكائن. بمعنى آخر، إذا أردت أن تعرف إن كان الكائن بطةً أم لا، فعليك التحقُّق مما إذا كان ذلك الكائن يمشي مشيَ البطة، وصوته كصوت البطة، بدلًا من أن تسأل عما إذا كان الكائن بطةً.

عندما تحتوي عدة أصناف أو أصناف فرعية على توابع لها نفس الأسماء، ولكن بسلوكيات مختلفة، نقول إنَّ تلك الأصناف متعدِّدة الأشكال (polymorphic) لأنَّها تستعمل واجهة موحدة يمكن استخدامها مع كيانات من أنواع مختلفة. يمكن للدوال تقييم ومعالجة هذه التوابع متعدِّدة الأشكال دون معرفة أصنافها.

2. إنشاء أصناف متعددة الأشكال

للاستفادة من التَعدُّدية الشكلية، سننشئ صنفين مختلفين لاستخدامهما مع كائنين مختلفين. يحتاج هذان الصنفان المختلفان إلى واجهة موحدة يمكن استخدامها بطريقة تعدُّدية الشكل (polymorphically)، لذلك سنعرِّف فيهما توابع مختلفة، ولكن لها نفس الاسم.

سننشئ صنفًا باسم `Shark` وصنفًا آخر باسم `Clownfish`، وسيُعرِّف كل منهما التوابع `swim()` و `swim_backwards()` و `skeleton()`.

```
class Shark():
   def swim(self):
     print("يسبح القرش.")

   def swim_backwards(self):
     print("لا يمكن للقرش أن يسبح إلى الوراء، لكن يمكنه أن يغوص 
إلى الوراء.")

   def skeleton(self):
     print("هيكل القرش مصنوع من الغضروف.")


class Clownfish():
   def swim(self):
     print("تسبح سمكة المهرج.")

   def swim_backwards(self):
     print("يمكن لسمكة المهرج أن تسبح إلى الخلف.")

   def skeleton(self):
     print("هيكل سمكة المهرج مصنوع من العظام.")
```

في الشيفرة أعلاه، لدى الصنفين Shark و Clownfish ثلاثة توابع تحمل نفس الاسم بيْد أنَّ وظائف تلك التوابع تختلف من صنف لآخر.

دعنا نستنسخ (instantiate) من هذين الصنفين كائنين:

```
...
sammy = Shark()
sammy.skeleton()

casey = Clownfish()
casey.skeleton()
```

عند تنفيذ البرنامج باستخدام الأمر python polymorphic_fish.py، يمكننا أن نرى أنَّ كل كائن يتصرف كما هو متوقع:

```
هيكل القرش مصنوع من الغضروف.
هيكل سمكة المهرج مصنوع من العظام.
```

الآن وقد أصبح لدينا كائنين يستخدمان نفس الواجهة، فبمقدورنا استخدام هذين الكائنين بنفس الطريقة بغض النظر عن نوعيهما.

3. التعددية الشكلية في توابع الأصناف

لإظهار كيف يمكن لبايثون استخدام الصنفين المختلفين اللذين عرَّفناهما أعلاه بنفس الطريقة، سننشئ أولاً حلقة for، والتي ستمر على صف من الكائنات. ثم سنستدعي التوابع بغض النظر عن نوع الصنف الذي ينتمي إليه كل كائن. إلا أنَّنا سنفترض أنَّ تلك التوابع موجودة في كل تلك الأصناف.

```
...
sammy = Shark()

casey = Clownfish()

for fish in (sammy, casey):
    fish.swim()
    fish.swim_backwards()
    fish.skeleton()
```

لدينا كائنان، sammy من الصنف Shark، و casey من الصنف Clownfish. تمر حلقة for

على هـذين الكـائنين، وتسـتدعي التوابـع `swim()` و `swim_backwards()` و `skeleton()` على كل منها.

عند تنفيذ البرنامج، سنحصل على المخرجات التالية:

```
القرش يسبح.
لا يمكن للقرش أن يسبح إلى الوراء، لكن يمكنه أن يغوص إلى الوراء.
هيكل القرش مصنوع من الغضروف.
سمكة المهرج تسبح.
يمكن لسمكة المهرج أن تسبح إلى الخلف.
هيكل سمكة المهرج مصنوع من العظام.
```

مرت الحلقة `for` على الكائن `sammy` من الصنف `Shark`، ثم على الكائن `casey` المنتمي إلى الصـنف `Clownfish`، لـذلك نـرى التوابـع الخاصـة بالصـنف `Shark` قبـل التوابـع الخاصـة بالصنف `Clownfish`.

يدلُّ هذا على أنَّ بايثون تستخدم هذه التوابع دون أن تعرف أو تعبأ بتحديد نوع الصنف الخاص بالكائنات. وهذا مثال حي على استخدام التوابع بطريقة مُتعدِّدَة الأشكال.

4. التعددية الشكلية في الدوال

يمكننا أيضًا إنشاء دالة تقبل أيَّ شيء، وهذا سيسمح باستخدام التعددية الشكلية.

لننشئ دالة تسمى `in_the_pacific()`، والتي تأخذ كائنًا يمكننا تسميته `fish`. رغم أنَّنا سنستخدم الاسم `fish`، إلا أنَّه يمكننا استدعاء أي كائن في هذه الدالة:

```
...
def in_the_pacific(fish):
```

بعـد ذلـك، سـنجعل الدالـة تسـتخدم الكـائن `fish` الـذي مرَّرنـاه إليهـا. وفي هـذه الحالـة،

سنستدعي التابع `swim()` المعرّف في كل من الصنفين `Shark` و `Clownfish`:

```
...
def in_the_pacific(fish):
    fish.swim()
```

بعد ذلك، سننشئ نسخًا (instantiations) من الصنفين `Shark` و `Clownfish` لنمرِّرهما بعد ذلك إلى نفس الدالة `in_the_pacific()`:

```
...
def in_the_pacific(fish):
    fish.swim()

sammy = Shark()

casey = Clownfish()

in_the_pacific(sammy)
in_the_pacific(casey)
```

عند تنفيذ البرنامج، سنحصل على المخرجات التالية:

```
القرش يسبح.
سمكة المهرج تسبح.
```

رغم أننا مرَّرنا كائنًا عشوائيًا (`fish`) إلى الدالة `in_the_pacific()` عند تعريفها، إلا أنَّنا ما زلنا قادرين على استخدامها استخدامًا فعالًا، وتمرير نسخ من الصنفين `Shark` و `Clownfish` إليها. استدعى الكائنُ `casey` التابعَ `swim()` المُعرَّف في الصنف `Clownfish`، فيما استدعى الكائنُ `sammy` التابعَ `swim()` المُعرَّف في الصنف `Shark`.

5. خلاصة الفصل

تسمح التعدُّدية الشكلية باستخدام الكائنات بغض النظر عن نوعها، وهذا يوفر لبايثون مرونة كبيرة، وقابلية لتوسيع الشيفرة الكائنية.

21

تنقيح الشيفرات: استخدام منقِّح بايثون

التنقيح (debugging) -في تطـوير البرمجيـات- هي عمليـة البحث عم حـل المشـاكل الـتي تمنع عمل البرمجيـة عملًا سليمًا. يـوفِّر منقِّح بايثون (Python Debugger) بيئة متكاملة لتنقيح بـرامج بـايثون، إذ تـدعم ضـبط مواضـع التوقـف (breakpoints) الشـرطية، وتنفيـذ الشـيفرات المصدرية سطرًا بسطر، وتفحص مكدِّس (stack) الاستدعاء، وخلاف ذلك.

1. تشغيل منقح بايثون تفاعليًا

يأتي منقِّح بايثون جزءًا من تثبيت بايثون القياسي بشكل وحدة باسم pdb. يمكن توسعة المنقِّح بمزيد من الوظائف، ويُعرَّف بالصنف Pdb. يمكنك العودة إلى التوثيق الرسمي للمنقح pdb لمزيـدٍ من المعلومـات. سـنبدأ تجاربنـا مـع برنـامجٍ قصـير يحتـوي على متغـيرين عـامين (global variables) ودالـة (function) الـتي تحتـوي على حلقـة تكـرار for متشـعِّبة، والبنيـة الشهيرة `if __name__ == '__main__':` التي تستدعي الدالة `nested_loop()`:

```
num_list = [500, 600, 700]
alpha_list = ['x', 'y', 'z']


def nested_loop():
    for number in num_list:
      print(number)
      for letter in alpha_list:
        print(letter)

if __name__ == '__main__':
    nested_loop()
```

يمكننا الآن تشغيل البرنامج باستخدام منقح بايثون عبر الأمر الآتي:

```
python -m pdb looping.py
```

سيؤدي استخدام خيار سطر الأوامر ‎-m إلى استيراد أي وحدة بايثون تريدها، وفي حالتنا فسنستورد الوحدة pdb، والتي سنمررها إلى الخيار ‎-m كما هو مبيِّن في الأمر السابق. ستحصل على الناتج الآتي بعد تنفيذك للأمر السابق:

```
> /Users/sammy/looping.py(1)<module>()
-> num_list = [500, 600, 700]
(Pdb)
```

لاحظنا في أول سطر من المخرجات احتواءه على اسم الوحدة التي تنفَّذ حاليًا (كما هو موضح بالكلمة <module> مع كامل مسار السكربت، ويليه رقم السطر التي نُفِّذ أول مرة (وفي هذه الحالة سيكون الرقم 1، لكن قد توجد تعليقات أو أسطر غير قابلة للتنفيذ في بداية الملف، لذا قد يكون الرقم أكبر في بعض الحالات).

يُظهِر السطر الثاني ما هو السطر الحالي الذي يُنفَّذ حاليًا، ولمَّا كنَّا قد شغلنا المنقح pdb تفاعليًا فسيوفر لنا سطر أوامر تفاعلي للتنقيح. ويمكنك كتابة الأمر help للتعرف على الأوامر الخاصة بالمنقح، و help command لمزيد من المعلومات حول أمرٍ معيَّن.

الحظ أن سطر أوامر pdb يختلف عن الوضع التفاعلي في بايثون، والحظ أنَّ منقح بايثون سيبدأ من جديد حين وصوله إلى نهاية البرنامج تلقائيًا؛ لذا يمكنك استخدام الأمر quit أو exit في أي وقتٍ تريد للخروج من المنقح. أمَّا إذا أردت إعادة تشغيل المنقح من بداية البرنامج مجددًا، فيمكنك استعمال الأمر run.

2. استخدام المنقح للتنقل ضمن البرنامج

من الشائع أثناء عملك مع منقح بايثون أن تستعمل الأوامر list و step و next للتحرك ضمن الشيفرة. سنشرح هذه الأوامر في هذا القسم. يمكننا كتابة الأمر list ضمن منقح بايثون التفاعلي للحصول على الشيفرات المحيطة بالسطر الحالي، فلو نفذناه عند السطر الأول من برنامج looping.py فستبدو المخرجات كما يلي:

```
(Pdb) list
  1  -> num_list = [500, 600, 700]
  2     alpha_list = ['x', 'y', 'z']
  3
  4
  5     def nested_loop():
  6       for number in num_list:
  7         print(number)
  8         for letter in alpha_list:
  9           print(letter)
 10
 11     if __name__ == '__main__':
(Pdb)
```

يُشار إلى السطر الحالي بالمحرفين <- اللذين يشيران في حالتنا إلى أول سطر من البرنامج. ولمَّا كان برنامجنا قصيرًا، فسنحصل على كامل البرنامج عند استخدام الأمر list. فحين استخدام الأمر list دون أيَّة وسائط، فسيعرض أحد عشر سطرًا محيطًا بالسطر الحالي، لكن يمكننا تحديد ما هي الأسطر التي نريد عرضها كما يلي:

```
(Pdb) list 3, 7
  3
  4
  5   def nested_loop():
  6     for number in num_list:
  7       print(number)
(Pdb)
```

طلبنا في المثال السابق أن يعرض المنقح الأسطر 3 إلى 7، وذلك باستخدام الأمر `list 3, 7`. للتنقل عبر البرنامج سطرًا بسطر، فيمكننا استخدام الأمر `step` أو `next`:

```
(Pdb) step
> /Users/sammy/looping.py(2)<module>()
-> alpha_list = ['x', 'y', 'z']
(Pdb)
(Pdb) next
> /Users/sammy/looping.py(2)<module>()
-> alpha_list = ['x', 'y', 'z']
(Pdb)
```

الفرق بين `step` و `next` هو أنَّ `step` ستتوقف داخل دالة جرى استدعاؤها، أما `next` فستنفِّذ الدوال وتتوقف في السطر التالي من تنفيذ الدالة. سنرى هذا الفرق رأي العين حين نتعامل مع الدالة الموجودة في برنامجنا. الأمر `step` سيمر على الحلقات خطوةً خطوة حتى يصل إلى نهاية الدالة، مما يُظهِر ما الذي تفعله الحلقة تمامًا، إذ سنبدأ بطباعة الرقم `print(number)` ثم ننطلق إلى طباعة الأحرف `print(letter)` ثم نعود إلى الرقم وهكذا.

```
(Pdb) step
> /Users/sammy/looping.py(5)<module>()
-> def nested_loop():
(Pdb) step
```

```
> /Users/sammy/looping.py(11)<module>()
-> if __name__ == '__main__':
(Pdb) step
> /Users/sammy/looping.py(12)<module>()
-> nested_loop()
(Pdb) step
--Call--
> /Users/sammy/looping.py(5)nested_loop()
-> def nested_loop():
(Pdb) step
> /Users/sammy/looping.py(6)nested_loop()
-> for number in num_list:
(Pdb) step
> /Users/sammy/looping.py(7)nested_loop()
-> print(number)
(Pdb) step
500
> /Users/sammy/looping.py(8)nested_loop()
-> for letter in alpha_list:
(Pdb) step
> /Users/sammy/looping.py(9)nested_loop()
-> print(letter)
(Pdb) step
x
> /Users/sammy/looping.py(8)nested_loop()
-> for letter in alpha_list:
(Pdb) step
> /Users/sammy/looping.py(9)nested_loop()
-> print(letter)
(Pdb) step
y
> /Users/sammy/looping.py(8)nested_loop()
-> for letter in alpha_list:
```

```
(Pdb)
```

أمَّا الأمر next فسينفِّذ الدالة بأكملها دون أن يرينا العملية خطوةً بخطوة. لنغلق الجلسة الحالية باستخدام الأمر exit ثم نُشغِّل المنقِّح مجددًا:

```
python -m pdb looping.py
```

يمكننا الآن تجربة الأمر next:

```
(Pdb) next
> /Users/sammy/looping.py(5)<module>()
-> def nested_loop():
(Pdb) next
> /Users/sammy/looping.py(11)<module>()
-> if __name__ == '__main__':
(Pdb) next
> /Users/sammy/looping.py(12)<module>()
-> nested_loop()
(Pdb) next
500
x
y
z
600
x
y
z
700
x
y
z
--Return--
> /Users/sammy/looping.py(12)<module>()->None
```

```
-> nested_loop()
(Pdb)
```

قد ترغب في تفحُّص القيم المُسنَدة إلى المتغيرات أثناء مرورك على الشيفرة، ويمكنك فعل ذلك باستخدام الأمر pp، والذي يطبع قيمة التعبير المُمرَّر إليه باستخدام الوحدة pprint:

```
(Pdb) pp num_list
[500, 600, 700]
(Pdb)
```

تملك أغلبية الأوامر في المنقح pdb اختصارات لها، فمثلًا الشكل المختصر من الأمر step هو s، والمختصر من next هو n. سيعرض لك الأمر help قائمة الاختصارات المتاحة. يمكنك أيضًا إعادة استدعاء آخر أمر نُفِّذ في المنقح بالضغط على زر الإدخال Enter.

3. نقاط التوقف

من المرجح أنَّك ستعمل على برمجيات أكبر بكثير من المثال السابق، لذا قد ترغب بتفحص دوال أو أسطر معينة بدلًا من المرور على كامل البرنامج، ويمكنك أن تضبط نقاط التوقف (breakpoints) باستخدام الأمر break، فسيعمل البرنامج حتى نقطة التوقف المحددة. عندما تضيف نقطة توقف فسيسند المنقح رقمًا إليها، وهذه الأرقام متتالية وتبدأ من 1، والتي يمكنك الاستفادة منها حين التعامل مع نقاط التوقف. يمكن إضافة نقاط توقف في أسطر برمجية معينة باستخدام الصيغة الآتية في منقح pdb على الشكل <program_file>:<line_number>:

```
(Pdb) break looping.py:5
Breakpoint 1 at /Users/sammy/looping.py:5
(Pdb)
```

اكتب clear ثم y لإزالة جميع نقاط التوقف الحالية، يمكنك بعدها أن تضع نقطة توقف

مكان تعريف الدالة:

```
(Pdb) break looping.nested_loop
Breakpoint 1 at /Users/sammy/looping.py:5
(Pdb)
```

لإزالة نقاط التوقف الحالية، اكتب clear ثم y مجددًا. يجدر بالذكر أنَّك تستطيع أن تضيف شرطًا:

```
(Pdb) break looping.py:7, number > 500
Breakpoint 1 at /Users/sammy/looping.py:7
(Pdb)
```

حينما نستعمل الآن الأمر continue فسيتوقف تنفيذ البرنامج عندما تكون قيمة الرقم أكبر من 500 (أي عندما يكون مساويًا إلى 600، وذلك في الدورة الثانية لحلقة التكرار الخارجية):

```
(Pdb) continue
500
x
y
z
> /Users/sammy/looping.py(7)nested_loop()
-> print(number)
(Pdb)
```

لرؤية قائمة من نقاط التوقف التي ضبطت، فيمكنك أن تستعمل الأمر break دون أي وسائط، وستحصل على معلومات حول نقاط التوقف جميعها التي ضبطتها:

```
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/sammy/looping.py:7
    stop only if number > 500
```

```
    breakpoint already hit 2 times
(Pdb)
```

يمكنك أيضًا تعطيل نقطة توقف باستخدام الأمر disable مع رقم نقطة التوقف. إذ سنضيف في هذا المثال نقطة توقف جديدة ثم نعطِّل أول نقطة توقف:

```
(Pdb) break looping.py:11
Breakpoint 2 at /Users/sammy/looping.py:11
(Pdb) disable 1
Disabled breakpoint 1 at /Users/sammy/looping.py:7
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep no    at /Users/sammy/looping.py:7
    stop only if number > 500
    breakpoint already hit 2 times
2   breakpoint   keep yes   at /Users/sammy/looping.py:11
(Pdb)
```

لتفعيل نقطة توقف، استخدم الأمر enable، ولإزالة نقطة التوقف كليًا فاستخدم الأمر clear:

```
(Pdb) enable 1
Enabled breakpoint 1 at /Users/sammy/looping.py:7
(Pdb) clear 2
Deleted breakpoint 2 at /Users/sammy/looping.py:11
(Pdb)
```

تمنحك نقاط التوقف في pdb تحكمًا دقيقًا في عملية التنقيح، وهنالك وظائف إضافية لها تتضمن تجاهل نقاط التوقف في الجلسة الحالية من البرنامج باستخدام الأمر ignore (كما في الأمر ignore 1)، وتشغيل إجراءات في نقاط التوقف باستخدام الأمر command (كما في الأمر

1 command)، وإنشاء نقاط توقف مؤقَّتة ستُحذَف تلقائيًا بعد الوصول إليها وذلك باستخدام الأمر tbreak (فلو أردنا إنشاء نقطة توقف مؤقتة في السطر الثالث مثلًا، نكتب tbreak 3).

4. دمج pdb مع البرامج

يمكنك أن تبدأ جلسة التنقيح باستيراد الوحدة pdb وإضافة الدالة pdb.set_trace() قبل السطر الذي تريد بدء جلسة التنقيح منه. إذ سنضيف في مثالنا السابق عبارة import ونبدأ عملية التنقيح داخل الدالة قبل حلقة التكرار الداخلية:

```
# استيراد الوحدة
import pdb

num_list = [500, 600, 700]
alpha_list = ['x', 'y', 'z']


def nested_loop():
    for number in num_list:
        print(number)

        # تشغيل المنقح هنا
        pdb.set_trace()
        for letter in alpha_list:
            print(letter)

if __name__ == '__main__':
    nested_loop()
```

بإضافة المنقِّح إلى شيفرتك، فلن تحتاج إلى تشغيل برنامجك بطريقة خاصة، أو تذكر ضبط نقاط التوقف. يسمح لك استيراد الوحدة pdb واستدعاء الدالة pdb.set_trace() ببدء

برنامج مثل المعتاد، وتشغيل المنقِّح أثناء تنفيذ البرنامج.

5. تعديل تسلسل تنفيذ البرنامج

يسمح لنا منقح بايثون بتغيير تسلسل تنفيذ البرنامج باستخدام الأمر `jump`، وهذا يسمح لك بالانتقال إلى الأمام في تنفيذ البرنامج لمنع شيفرة معيَّنة، أو يمكنك العودة إلى الخلف وتنفيذ الشيفرة مرةً أخرى. سنعمل هنا مع برنامج بسيط يُنشِئ قائمةً `list` من الحروف الموجودة في المتغير `sammy = "sammy"`:

```
def print_sammy():
    sammy_list = []
    sammy = "sammy"
    for letter in sammy:
        sammy_list.append(letter)
        print(sammy_list)

if __name__ == "__main__":
    print_sammy()
```

إذا شغلنا البرنامج بالشكل المعتاد باستخدام الأمر `python letter_list.py` فسنحصل على الناتج الآتي:

```
['s']
['s', 'a']
['s', 'a', 'm']
['s', 'a', 'm', 'm']
['s', 'a', 'm', 'm', 'y']
```

أمَّا مع منقح بايثون، فسنرى كيف يمكننا تغيير ترتيب التنفيذ بتخطي أول دورة من تنفيذ حلقة التكرار، وعندما نفعل ذلك، فسنلاحظ كيف تغيَّر ترتيب تنفيذ الشيفرة:

```
python -m pdb letter_list.py
> /Users/sammy/letter_list.py(1)<module>()
-> def print_sammy():
(Pdb) list
  1  -> def print_sammy():
  2         sammy_list = []
  3         sammy = "sammy"
  4         for letter in sammy:
  5             sammy_list.append(letter)
  6             print(sammy_list)
  7
  8     if __name__ == "__main__":
  9         print_sammy()
 10
 11
(Pdb) break 5
Breakpoint 1 at /Users/sammy/letter_list.py:5
(Pdb) continue
> /Users/sammy/letter_list.py(5)print_sammy()
-> sammy_list.append(letter)
(Pdb) pp letter
's'
(Pdb) continue
['s']
> /Users/sammy/letter_list.py(5)print_sammy()
-> sammy_list.append(letter)
(Pdb) jump 6
> /Users/sammy/letter_list.py(6)print_sammy()
-> print(sammy_list)
(Pdb) pp letter
'a'
(Pdb) disable 1
Disabled breakpoint 1 at /Users/sammy/letter_list.py:5
```

```
(Pdb) continue
['s']
['s', 'm']
['s', 'm', 'm']
['s', 'm', 'm', 'y']
```

جلسة التنقيح السابقة تضع نقطة توقف في السطر الخامس لمنع المنقح من تنفيذ الشيفرة التي تلي هذه النقطة، ثم تكمل تنفيذ الشيفرة (مع طباعة قيمة `letter` لنرى ما الذي يحدث)، ثمَّ سنستخدم الأمر `jump` للتخطي إلى السطر السادس، وفي هذه النقطة كانت قيمة المتغير `letter` تساوي السلسلة النصية `'a'`، لكننا لمَّا تجاوزنا الشيفرة، فلن تضاف هذه القيمة إلى القائمة (list) المسماة `sammy_list`، ومن بعدها عطلنا نقطة التوقف للاستمرار في تنفيذ البرنامج تنفيذًا طبيعيًا باستخدام الأمر `continue`، وكانت النتيجة هي عدم إضافة الحرف `'a'` إلى القائمة `sammy_list`.

يمكننا الآن إعادة تشغيل المنقح للعودة إلى البرنامج وإعادة تشغيل الأمر البرمجية التي جرى تنفيذه مسبقًا، وفي هذه المرة سنشغل أول حلقة تكرار في `for` في المنقِّح:

```
> /Users/sammy/letter_list.py(1)<module>()
-> def print_sammy():
(Pdb) list
  1  -> def print_sammy():
  2         sammy_list = []
  3         sammy = "sammy"
  4         for letter in sammy:
  5             sammy_list.append(letter)
  6             print(sammy_list)
  7
  8     if __name__ == "__main__":
```

```
  9         print_sammy()
 10
 11
(Pdb) break 6
Breakpoint 1 at /Users/sammy/letter_list.py:6
(Pdb) continue
> /Users/sammy/letter_list.py(6)print_sammy()
-> print(sammy_list)
(Pdb) pp letter
's'
(Pdb) jump 5
> /Users/sammy/letter_list.py(5)print_sammy()
-> sammy_list.append(letter)
(Pdb) continue
> /Users/sammy/letter_list.py(6)print_sammy()
-> print(sammy_list)
(Pdb) pp letter
's'
(Pdb) disable 1
Disabled breakpoint 1 at /Users/sammy/letter_list.py:6
(Pdb) continue
['s', 's']
['s', 's', 'a']
['s', 's', 'a', 'm']
['s', 's', 'a', 'm', 'm']
['s', 's', 'a', 'm', 'm', 'y']
```

أضفنا في جلسة التنقيح السابقة نقطة توقف في السطر السادس، ثم «قفزنا» إلى السطر الخامس بعد الإكمال، ورأينا أن السلسلة النصية `'s'` قد أضيفت مرتين إلى القائمة `sammy_list`، ثم عطّلنا نقطة التوقف في السطر السادس وأكملنا تنفيذ البرنامج، ورأينا في المخرجات وجود حرفَي `'s'` مضافين إلى القائمة `sammy_list`.

يمكن أن يمنع المنقِّح بعض أنواع القفزات، مثل القفز داخل وخارج بنية تحكم، فمثلًا لا يمكنك أن تقفز إلى تنفيذ دالة قبل تعريف وسائطها، ولا يمكنك أن تقفز إلى داخل عبارة `try:except`. ولا يمكنك أيضًا أن تقفز خارج بنية `finally`. تسمح لنا عبارة `jump` الموجودة في منقِّح بايثون بتغيير تسلسل تنفيذ البرنامج أثناء تنقيحه لنرى إن كان بالإمكان تحسين هذا الجزء أو فهم سبب مشكلة معينة في الشيفرة.

6. جدول بأوامر pdb الشائعة

الجدول التالي فيه أوامر pdb المفيدة مع اختصاراتها لتبقيها في ذهنك أثناء تعاملك مع منقح بايثون:

الأمر	الاختصار	الوظيفة
args	a	طباعة قائمة الوسائط للدالة الحالية.
break	b	إنشاء نقطة توقف أثناء تنفيذ البرنامج (يتطلب وسيط)
continue	c أو cont	إكمال تنفيذ البرنامج.
help	h	توفير قائمة الأوامر، أو توفير مساعدة لأمر معين.
jump	j	ضبط ما هو السطر القادم الذي يجب تنفيذه.
list	l	طباعة الشيفرة المصدرية المحيطة بالسطر الحالي.

الأمر	الاختصار	الوظيفة
next	n	إكمال التنفيذ حتى السطر التالي في الدالة، أو الخروج منها.
step	s	تنفيذ السطر الحالي، والتوقف في أول فرصة ممكنة.
pp	pp	طباعة قيمة التعبير.
quit أو exit	q	الخروج من البرنامج.
return	r	إكمال التنفيذ حتى تعيد الدالة قيمةً ما.

يمكنك قراءة المزيد عن الأوامر السابقة والتعامل مع المنقح عبر توثيق بايثون الرسمي.

7. الوحدة code: تنقيح الشيفرات من سطر الأوامر التفاعلي

الوحدة code هي إحدى الأدوات المفيدة التي يمكن استخدامها لمحاكاة المترجم التفاعلي (interpreter)، إذ توفر هذه الوحدة فرصةً لتجربة الشيفرة التي تكتبها.

بدلاً من تفحُّص الشيفرة باستخدام منقِّح، يمكنك إضافة الوحدة code لوضع نقاط لإيقاف تنفيذ البرنامج، والدخول في الوضع التفاعلي لتفحُّص ومتابعة كيفيَّة عمل الشيفرة. الوحدة code هي جزء من مكتبة بايثون القياسية.

هذه الوحدة مفيدةٌ لأنَّها ستمكنك من استخدام مترجم دون التضحية بالتعقيد والاستدامة التي توفرها ملفات البرمجة. فيمكنك عبر استخدام الوحدة code تجنب استخدام الدالة print() في شيفرتك لأجل التنقيح، لأنَّها طريقة غير عمليَّة. لاستخدامها في تنقيح

الأخطاء، يمكنك استخدام الدالة `interact()` الخاصَّة بالوحدة `code`، والتي توقف تنفيذ البرنامج عند استدعائها، وتوفر لك سطر أوامر تفاعلي حتى تتمكن من فحص الوضع الحالي لبرنامجك.

تُكتَب الدالة `interact()` بالشكل التالي:

```
code.interact(banner=None, readfunc=None, local=None,
exitmsg=None)
```

تُنفِّذ هذه الدالة حلقة اقرأ-قيِّم-اطبع (تختصر إلى REPL، أي Read–eval–print loop، وتنشئ نسخة من الصنف `InteractiveConsole`، والذي يحاكي سلوك مترجم بايثون التفاعلي.

هذه هي المعاملات الاختيارية:

- `banner`: يمكن أن تعطيه سلسلة نصية لتعيين موضع إطلاق المترجم.
- `readfunc`: يمكن استخدامه مثل التابع `InteractiveConsole.raw_input()`.
- `local`: سيعيِّن فضاء الأسماء (namespace) الافتراضي لحلقة المترجم (interpreter loop).
- `exitmsg`: يمكن إعطاؤه سلسلة نصية لتعيين موضع توقف المترجم.

مثلًا، يمكن استخدام المعامل `local` بهذا الشكل:

- `local=locals()` - لفضاء أسماء محلي
- `local=globals()` - لفضاء أسماء عام

- `local=dict(globals(), **locals())` - لاستخدام كل من فضاء الأسماء العام، وفضاء الأسماء المحلي الحالي

المعامل `exitmsg` جديد، ولم يظهر حتى إصدار بايثون 3.6، لذلك إن كنت تستخدم إصدارًا أقدم، فحدِّثه، أو لا تستخدم المعامل `exitmsg`.

ضع الدالة `interact()` حيث تريد إطلاق المترجم التفاعلي في الشيفرة.

أ. كيفية استخدام الوحدة code

لتوضيح كيفية استخدام الوحدة `code`، سنكتب بُريمجًا عن الحسابات المصرفيّة يسمى `balances.py`. سنعيّن المعامل المحلي عند القيمة `locals()` لجعل فضاء الأسماء محليًّا.

```
# code استيراد الوحدة
import code

bal_a = 2324
bal_b = 0
bal_c = 409
bal_d = -2

account_balances = [bal_a, bal_b, bal_c, bal_d]


def display_bal():
    for balance in account_balances:
        if balance < 0:
            print("Account balance of {} is below 0; add funds now."
                .format(balance))

        elif balance == 0:
```

```
        print("Account balance of {} is equal to 0; add funds
soon."
             .format(balance))

      else:
        print("Account balance of {} is above
0.".format(balance))

# استخدام interact() لبدء المترجم بفضاء أسماء محلي
code.interact(local=locals())

display_bal()
```

لقد استدعينا الدالة `code.interact()` مع المعامل `local=locals()` لاستخدام فضاء الأسماء المحلي بوصفه قيمة افتراضية داخل حلقة المترجم.

لتُنفِّذ البرنامج أعلاه باستخدام الأمر `python3` إذا لم نكن تعمل في بيئة افتراضية، أو الأمر `python` خلاف ذلك:

```
python balances.py
```

بمجرَّد تنفيذ البرنامج، سنحصل على المخرجات التالية:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more
information.
(InteractiveConsole)
>>>
```

سيُوضَع المؤشر في نهاية السطر `>>>`، كما لو أنَّك في سطر الأوامر التفاعلي. من هنا، يمكنك استدعاء الدالة `print()` لطباعة المتغيرات والدوال وغير ذلك:

```
>>> print(bal_c)
409
>>> print(account_balances)
[2324, 0, 409, -2]
>>> print(display_bal())
Account balance of 2324 is 0 or above.
Account balance of 0 is equal to 0, add funds soon.
Account balance of 409 is 0 or above.
Account balance of -2 is below 0, add funds now.
None
>>> print(display_bal)
<function display_bal at 0x104b80f28>
```

نرى أنَّه باستخدام فضاء الأسماء المحلي، يمكننا طباعة المتغيرات، واستدعاء الدالة. يُظهر الاستدعاء الأخير للدالة `print()` أنّ الدالة `display_bal` موجودة في ذاكرة الحاسوب.

بعد أن تنتهي من العمل على المترجم، يمكنك الضغط على CTRL + D في الأنظمة المستندة إلى يونكس، أو CTRL + Z في أنظمة ويندوز لمغادرة سطر الأوامر ومتابعة تنفيذ البرنامج.

إذا أردت الخروج من سطر الأوامر دون تنفيذ الجزء المتبقي من البرنامج، فاكتب `quit()`، وسيتوقف البرنامج.

في المثال التالي، سنستخدم المُعاملين `banner` و `exitmsg`:

```
# استخدم الدالة interact() لبدء المترجم
code.interact(banner="Start", local=locals(), exitmsg="End")

display_bal()
```

عند تنفيذ البرنامج، ستحصل على المخرجات التالية:

```
Start
```

```
>>>
```

يـتيح لـك اسـتخدام المعامـل banner تعـيين عـدّة نقـاط داخـل شـيفرتك، مـع القـدرة على تحديـدها. على سـبيل المثـال، يمكن أن يكـون لـديك معامـل banner يطبـع السلسـلة النصـية "In for-loop" مع معامل exmsg يطبع "Out of for-loop"، وذلك حتى تعرف مكانك بالضبط في الشيفرة.

من هنا، يمكننا استخدام المترجم مثل المعتاد. بعد كتابة CTRL + D للخروج من المترجم، ستحصل على رسالة الخروج، وسيتم تنفيذ الدالة:

```
End
Account balance of 2324 is 0 or above.
Account balance of 0 is equal to 0, add funds soon.
Account balance of 409 is 0 or above.
Account balance of -2 is below 0, add funds now.
```

سيتم تنفيذ البرنامج بالكامل بعد الجلسة التفاعلية.

بمجرد الانتهاء من استخدام الوحدة code لتنقيح الشيفرة، يجب عليك إزالة دوال الوحدة code وعبارة الاستيراد حتى يُنفَّذ البرنامج مثل المعتاد.

8. الوحدة Logging: التنقيح بالتسجيل وتتبع الأحداث

الوحـدة logging هي جـزء من مكتبـة بـايثون القياسـية والـتي تـوفر تتبعًا للأحـداث الـتي تحصل أثناء تشغيل البرنامج، ويمكننا إضافة استدعاءات للتسجيل ضمن الشيفرة للإشارة إلى حـدوث أمـر معيَّن. تسـمح الوحـدة logging بالتسـجيل لأغـراض استكشـاف المشـاكل وإصـلاحها، وتسجيل الأحداث المتعلقة بتشغيل التطبيق، إضافةً إلى سجل الأحداث الذي يسجِّل تفاعلات

المستخدم لتحليلها. وتستعمل الوحدة خصوصًا لتسجيل الأحداث إلى ملف.

تبقي الوحدة logging سجلًا بالأحداث التي وقعت ضمن البرنامج، مما يسمح برؤية المخرجات المتعلقة بأي حدث التي تحدث أثناء تشغيل البرنامج. قد تكون معتادًا على التحقُّق من الأحداث باستخدام الدالة print في شيفرتك، وصحيحٌ أنَّ الدالة print توفر طريقة أساسية لمحاولة تنقيح الشيفرة وحل المشكلات. لكن استخدام print لتنقيح البرنامج وتتبع عملية التنفيذ وحالة التطبيق هو خيار صعبةٌ صيانته موازنةً مع الوحدة logging لعدّة أسباب:

- سيصبح صعبًا التفريق بين مخرجات التنقيح والمخرجات العادية للبرنامج، فستختلط المخرجات مع بعضها.
- لا توجد طريقة سهلة لتعطيل الدوال print عند استخدامها متناثرة في مواضع مختلفة في الشيفرة.
- يصعب حذف جميع دوال print عند الانتهاء من التنقيح.
- لا يوجد سجل يوضح ما هي معلومات التشخيص واستكشاف لأخطاء بموثوقيَّة عالية.

لذا من الأفضل أن نعتاد على استخدام الوحدة logging في الشيفرة لأنَّها أفضل للتطبيقات التي تتعدى كونها سكربت بايثون قصير، وتوفر طريقة أنسب للتنقيح. ولأنَّ السجلات تعرض سلوك وأخطاء التطبيق على فترة من الزمن، فستحصل على صورة شاملة عمَّا يحدث في عملية تطوير تطبيقك.

ا. طباعة رسائل التنقيح إلى الطرفية

إذا كنتَ معتادًا على استخدام الدالة print لترى ماذا يحدث في تطبيقك، فستكون معتادًا على رؤية تطبيق مثل المثال الآتي الذي يُعرِّف صنفًا ويُهيِّئ الكائنات فيه:

```
class Pizza():
    def __init__(self, name, price):
        self.name = name
        self.price = price
        print("Pizza created: {} (${})".format(self.name,
self.price))

    def make(self, quantity=1):
        print("Made {} {} pizza(s)".format(quantity, self.name))

    def eat(self, quantity=1):
        print("Ate {} pizza(s)".format(quantity, self.name))

pizza_01 = Pizza("artichoke", 15)
pizza_01.make()
pizza_01.eat()

pizza_02 = Pizza("margherita", 12)
pizza_02.make(2)
pizza_02.eat()
```

تحتوي الشيفرة السابقة على التابع `__init__` الذي يُعرِّف المعاملين `name` و `price` لكائن من الصنف `Pizza`. ولدى الصنف تابعين، أحدهما باسم `make()` لإنشاء البيتزا، والآخر باسم `eat()` لأكل البيتزا p-: هذان التابعان يقبلان معاملًا باسم `quantity` بقيمة ابتدائية تساوي.

لنشغِّل البرنامج:

```
python pizza.py
```

سنحصل على المخرجات الآتية:

```
Pizza created: artichoke ($15)
```

```
Made 1 artichoke pizza(s)
Ate 1 pizza(s)
Pizza created: margherita ($12)
Made 2 margherita pizza(s)
Ate 1 pizza(s)
```

صحيحٌ أننا نستخدم الدالة `print` لرؤية كيف تعمل الشيفرة، لكن يمكننا استخدام الوحدة `logging` لفعل ذلك بدلًا منها. لنزل الدالة `print` الموجودة في الشيفرة، ونضع `import logging` في بداية الملف:

```
import logging


class Pizza():
    def __init__(self, name, value):
        self.name = name
        self.value = value
...
```

المستوى الافتراضي للتسجيل في وحدة `logging` هو `WARNING` (تحذير) والذي هو مستوى أعلى بدرجة واحدة من `DEBUG` (تنقيح)، ولمّا كنّا نريد استخدام الوحدة `logging` للتنقيح في هذا المثال، فعلينا تغيير الضبط لكي يكون مستوى التسجيل هو `logging.DEBUG`، وذلك بإضافة السطر الآتي بعد عبارة الاستيراد:

```
import logging

logging.basicConfig(level=logging.DEBUG)


class Pizza():
```

```
...
```

تشير القيمة logging.DEBUG إلى قيمة رقمية ثابتة، وقيمة المستوى DEBUG هي 10.

لنبدِّل الآن جميع الدوال print إلى التابع logging.debug()، وعلى النقيض من logging.DEBUG التي هي قيمة عددية ثابتة، فإنَّ التابع logging.debug() خاصٌ بالوحدة logging، وعند التعامل مع هذا التابع فيمكننا استخدام نفس السلسلة النصية المُمرَّرة إلى print كما هو موضح أدناه:

```
import logging

logging.basicConfig(level=logging.DEBUG)


class Pizza():
   def __init__(self, name, price):
      self.name = name
      self.price = price
      logging.debug("Pizza created: {} (${})".format(self.name,
self.price))

   def make(self, quantity=1):
      logging.debug("Made {} {} pizza(s)".format(quantity,
self.name))

   def eat(self, quantity=1):
      logging.debug("Ate {} pizza(s)".format(quantity,
self.name))

pizza_01 = Pizza("artichoke", 15)
pizza_01.make()
```

```
pizza_01.eat()

pizza_02 = Pizza("margherita", 12)
pizza_02.make(2)
pizza_02.eat()
```

عندما نشغل البرنامج باستخدام الأمر `python pizza.py` فسنحصل على الناتج الآتي:

```
DEBUG:root:Pizza created: artichoke ($15)
DEBUG:root:Made 1 artichoke pizza(s)
DEBUG:root:Ate 1 pizza(s)
DEBUG:root:Pizza created: margherita ($12)
DEBUG:root:Made 2 margherita pizza(s)
DEBUG:root:Ate 1 pizza(s)
```

تملك رسائل التسجيل المستوى الأمني `DEBUG` إضافةً إلى الكلمة `root`، والتي تشير إلى مستوى وحدة بايثون الخاصة بك، إذ يمكن استخدام الوحدة `logging` مع هيكلية من المسجلات التي لها أسماء مختلفة، لذا يمكنك استخدام مسجِّل (logger) مختلف لكل وحدة من وحدات بايثون الخاصة بك. على سبيل المثال، يمكننا إسناد أكثر من مسجِّل معًا وإعطائها أسماء مختلفة:

```
logger1 = logging.getLogger("module_1")
logger2 = logging.getLogger("module_2")

logger1.debug("Module 1 debugger")
logger2.debug("Module 2 debugger")
DEBUG:module_1:Module 1 debugger
DEBUG:module_2:Module 2 debugger
```

بعد أن فهمنا كيفيَّة استخدام الوحدة logging لطباعة الرسائل إلى الطرفية، فلننتقل إلى استخدام الوحدة logging لطباعة الرسائل إلى ملف.

ب. تسجيل الرسائل إلى ملف

الهدف الرئيسي من الوحدة logging هي تسجيل الرسائل إلى ملف بدلًا من طباعتها إلى الطرفية، إذ يؤدي وجود ملف يحتوي على البيانات المُخزَّنة على فترة زمنية طويلة إلى إحصاء وتقدير ما هي التغييرات التي يجب إجراؤها على الشيفرة أو البرنامج ككل. يمكننا تعديل التابع logging.basicConfig() لبدء التسجيل إلى ملف، وذلك بتمرير المعامل filename، وفي هذه الحالة سندعو الملف باسم test.log:

```
import logging

logging.basicConfig(filename="test.log", level=logging.DEBUG)


class Pizza():
    def __init__(self, name, price):
        self.name = name
        self.price = price
        logging.debug("Pizza created: {} ($
{})".format(self.name, self.price))

    def make(self, quantity=1):
        logging.debug("Made {} {} pizza(s)".format(quantity,
self.name))

    def eat(self, quantity=1):
        logging.debug("Ate {} pizza(s)".format(quantity,
```

```
self.name))

pizza_01 = Pizza("artichoke", 15)
pizza_01.make()
pizza_01.eat()

pizza_02 = Pizza("margherita", 12)
pizza_02.make(2)
pizza_02.eat()
```

الشيفرة السابقة مشابهة كثيرًا للشيفرة الموجودة في القسم السابق، باستثناء أننا أضفنا اسم الملف `filename` لتخزين مخرجات السجل. وبعد تشغيل السكربت باستخدام الأمر `python pizza.py` فمن المفترض أن يُنشأ ملفٌ جديدٌ في المجلد الخاص بنا باسم `test.log`.

لنفتح الملف `test.log` باستخدام vi (أو أي محرر تفضله):

```
vi test.log
```

عند تفحُّص محتويات الملف، فسنرى ما يلي:

```
DEBUG:root:Pizza created: artichoke ($15)
DEBUG:root:Made 1 artichoke pizza(s)
DEBUG:root:Ate 1 pizza(s)
DEBUG:root:Pizza created: margherita ($12)
DEBUG:root:Made 2 margherita pizza(s)
DEBUG:root:Ate 1 pizza(s)
```

الناتج شبيه بمحتويات الطرفية التي رأيناها في القسم السابق، لكنَّها مُخزَّنة الآن في ملف `test.log`. لنعد إلى تعديل الملف `pizza.py` لتعديل الشيفرة، سنبقي أغلبية الشيفرة على حالتها، لكننا سنعدل معاملين في كائني `pizza_01` و `pizza_02`:

```
import logging

logging.basicConfig(filename="test.log", level=logging.DEBUG)


class Pizza():
    def __init__(self, name, price):
        self.name = name
        self.price = price
        logging.debug("Pizza created: {} ($
{})".format(self.name, self.price))

    def make(self, quantity=1):
        logging.debug("Made {} {} pizza(s)".format(quantity,
self.name))

    def eat(self, quantity=1):
        logging.debug("Ate {} pizza(s)".format(quantity,
self.name))

# تعديل معاملات الكائن
pizza_01 = Pizza("Sicilian", 18)
pizza_01.make(5)
pizza_01.eat(4)

# تعديل معاملات الكائن
pizza_02 = Pizza("quattro formaggi", 16)
pizza_02.make(2)
pizza_02.eat(2)
```

بعد حفظ التعديلات السابقة، لنعد تشغيل البرنامج بالأمر `python pizza.py`. بعد تشغيل البرنامج، لنستعرض محتوى الملف `test.log` بالمُحرِّر المفضل لديك:

```
vi test.log
```

عندما ننظر إلى الملف، فسنرى أنَّ هنالك أسطر جديدة قد أضيفت، وأنَّ الأسطر السابقة من المرة الماضيَّة ما تزال موجودةً:

```
DEBUG:root:Pizza created: artichoke ($15)
DEBUG:root:Made 1 artichoke pizza(s)
DEBUG:root:Ate 1 pizza(s)
DEBUG:root:Pizza created: margherita ($12)
DEBUG:root:Made 2 margherita pizza(s)
DEBUG:root:Ate 1 pizza(s)
DEBUG:root:Pizza created: Sicilian ($18)
DEBUG:root:Made 5 Sicilian pizza(s)
DEBUG:root:Ate 4 pizza(s)
DEBUG:root:Pizza created: quattro formaggi ($16)
DEBUG:root:Made 2 quattro formaggi pizza(s)
DEBUG:root:Ate 2 pizza(s)
```

وصحيحٌ أنَّ هذه المعلومات مفيدة بكل تأكيد، لكن يمكننا أن نجعل السجل مليئًا بالمعلومات بإضافة خاصية `LogRecord`. وأهم ما نريد فعله هو إضافة بصمة وقت قابلة للقراءة بسهولة التي تخبرنا متى أُنشِئ السجل. سنضيف ذلك إلى المعامل `format`، إذا نضع `%(asctime)s` لإضافة الوقت، ولإبقاء اسم المستوى (DEBUG) فيجب تضمين السلسلة النصية `%(levelname)s`، وللإبقاء على الرسائل التي نطلب من المسجل أن يسجلها، فعلينا تضمين `%(message)s`، كل سلسلة نصية من الخاصيات السابقة مفصولة عن بعضها بنقطتين رأسيتين : كما هو ظاهر في الشيفرة أدناه:

```
import logging

logging.basicConfig(
    filename="test.log",
    level=logging.DEBUG,
    format="%(asctime)s:%(levelname)s:%(message)s"
    )


class Pizza():
    def __init__(self, name, price):
        self.name = name
        self.price = price
        logging.debug("Pizza created: {} ($
{})".format(self.name, self.price))

    def make(self, quantity=1):
        logging.debug("Made {} {} pizza(s)".format(quantity,
self.name))

    def eat(self, quantity=1):
        logging.debug("Ate {} pizza(s)".format(quantity,
self.name))

pizza_01 = Pizza("Sicilian", 18)
pizza_01.make(5)
pizza_01.eat(4)

pizza_02 = Pizza("quattro formaggi", 16)
pizza_02.make(2)
pizza_02.eat(2)
```

عند تشغيل الشيفرة السابقة باستخدام الأمر `python pizza.py` فستحصل على أسطر

جديدة في الملف test.log التي تتضمن بصمة الوقت ومستوى التسجيل (DEBUG) والرسائل المرتبطة بها:

```
DEBUG:root:Pizza created: Sicilian ($18)
DEBUG:root:Made 5 Sicilian pizza(s)
DEBUG:root:Ate 4 pizza(s)
DEBUG:root:Pizza created: quattro formaggi ($16)
DEBUG:root:Made 2 quattro formaggi pizza(s)
DEBUG:root:Ate 2 pizza(s)
2017-05-01 16:28:54,593:DEBUG:Pizza created: Sicilian ($18)
2017-05-01 16:28:54,593:DEBUG:Made 5 Sicilian pizza(s)
2017-05-01 16:28:54,593:DEBUG:Ate 4 pizza(s)
2017-05-01 16:28:54,593:DEBUG:Pizza created: quattro formaggi
($16)
2017-05-01 16:28:54,593:DEBUG:Made 2 quattro formaggi pizza(s)
2017-05-01 16:28:54,593:DEBUG:Ate 2 pizza(s)
```

اعتمادًا على احتياجاتك، ربما تستعمل خاصيات LogRecord في شيفراتك لتخصيصها. تسجيل رسائل التنقيح وغيرها في ملفات منفصلة يسهِّل عليك فهم تطبيقك فهمًا كليًا على مرور الزمن، مما يعطيك فرصةً لتصحيح وتعديل الشيفرات ببصيرة.

ج. جدول بمستويات التسجيل

يمكنك إسناد مستوى أهمية إلى الحدث باستخدام مستويات التسجيل. مستويات التسجيل هي قيم عددية (ثابتة)، والتي تكون من مضاعفات العدد 10، بدءًا من المستوى NOTSET الذي يهيئ المسجِّل بقيمة عددية تساوي 0. يمكنك أيضًا تعريف المستويات الخاصة بك، لكن إذا عرفتَ مستوى بقيمة عددية مساوية للقيمة العددية لمستوى موجود مسبقًا، فستعيد كتابة الاسم المرتبط بتلك القيمة.

يُظهِر الجدول الآتي مختلف مستويات التسجيل مع القيم العددية المرتبطة بها، وما هي الدالة التي يمكن استعمالها لاستدعاء المستوى، ولأي مستوى من الرسائل تستخدم:

المستوى	القيمة العددية	الدالة	الغرض
CRITICAL	50	`logging.critical()`	عرض خطأ جاد، وقد لا يكون البرنامج قابلًا للاستخدام بعده.
ERROR	40	`logging.error()`	عرض خطأ جاد.
WARNING	30	`logging.warning()`	الإشارة أن شيئًا غير متوقع قد حدث أو قد يحدث.
INFO	20	`logging.info()`	التأكيد أن الأمور تسير على ما يرام.
DEBUG	10	`logging.debug()`	عرض معلومات تنقيحية.

تضبط وحدة `logging` المستوى الافتراضي إلى `WARNING`، لذا ستُسجَّل رسائل `WARNING` و `ERROR` و `CRITICAL` افتراضيًا. ففي المثال الآتي سنعدل الضبط للإشارة لتضمين مستوى `DEBUG`:

```
logging.basicConfig(level=logging.DEBUG)
```

يمكنك التعرُّف على المزيد من الأوامر والتعامل معها بالاطلاع على توثيق logging الرسمي.

9. خلاصة الفصل

عمليـة التنقيح هي خطـوة مهمـة في جميـع مشـاريع التطـوير البرمجيـة. ويـوفر لنـا منقِّح بايثون pdb بيئة تنقيح تفاعلية يمكن استخدامها مع أي برنامج بايثون نكتبه. باستفادتنا للميزات التي تسمح لنا بتوقف عمل البرنامج مؤقتًا، وإلقاء نظرة على المتغيرات، وإكمال تنفيذ البرنامج خطوةً بخطوة، مما يمكننا من فهم ماذا يفعل البرنامج بالتفصيل ويساعدنا على اكتشاف العلل وإصلاح المشاكل المنطقية.

تُستخدَم الوحـدة code لإطلاق سـطر الأوامـر التفـاعلي لتفحُّص الشـيفرة خطـوةً بخطـوة بقصـد فهم سـلوكها، وتعـديل الشـيفرة إن لـزم الأمـر. لقـراءة المزيـد حـول هـذا الموضـوع، يمكنـك مطالعة التوثيق الرسمي للوحدة code.

تساعد الوحدة logging -التي هي جزء من مكتبة بايثون القياسية- بتتبع الأحداث التي تحصل أثناء تشغيل البرنامج، ويمكن إخراج هذه الأحداث إلى ملفات منفصلة للسماح بتتبع ما يحدث عندما تعمل الشيفرة. وهذا يوفر لنا الفرصة لتنقيح الشيفرة اعتمادًا على فهمنا لمختلف الأحداث التي تطرأ أثناء تشغيل البرنامج على فترةٍ من الزمن.

22

إصدارات بايثون: بايثون 3 مقابل بايثون 2

قبل أن ننظر إلى إمكانيات إصدارَي بايثون 2 وبايثون 3 (مع الاختلافات البرمجية الرئيسية بينهما)، فلننظر إلى لمحة تاريخية عن الإصدارات الرئيسية الحديثة من بايثون.

1. بايثون 2

نُشِرَ هذا الإصدار في أواخر عام 2000، وأصبحت بايثون 2 لغة برمجة شاملة موازنةً بالإصدارات التي تسبقها وذلك بعد تطبيق اقتراح PEP (Python Enhancement Proposal)، وهو مواصفةٌ (specification) تقنيةٌ توفِّر معلومات إلى أعضاء مجتمع بايثون أو تصف ميزاتٍ جديدة في اللغة. بالإضافة إلى ذلك، تضمنت بايثون 2 ميزاتٍ برمجية جديدة مثل "cycle-detecting garbage collector" لأتمتة عملية إدارة الذاكرة، وزيادة دعم يونيكود لتدعم اللغة جميع المحارف المعيارية ...إلخ. وأثناء عملية تطوير بايثون 2 أضيفت ميزات جديدة بما في ذلك توحيد الأنواع والأصناف في بايثون في بنية هيكلية وحيدة (وذلك في إصدار 2.2 من بايثون).

2. بايثون 3

تعدّ بايثون 3 مستقبل لغة بايثون وهي قيد التطوير من اللغة، وهذا إصدارٌ رئيسيٌ نُشِر في أواخر عام 2008 لإصلاح بعض المشاكل الجوهرية في تصميم الإصدارات السابقة من اللغة، وكان التركيز أثناء تطوير بايثون 3 هو تحسين الشيفرات التي تبنى عليها اللغة وحذف التكرارات، مما يعني أنَّ هنالك طريقة وحيدة فقط لإنجاز مهمَّة معيَّنة.

التعديلات الأساسية التي حدثت في بايثون 3.0 تتضمن تغيير التعليمة `print` إلى دالة مُضمَّنة باللغة، وتحسين قسمة الأعداد الصحيحة، وتوفير دعم إضافي ليونيكود.

في البداية، انتشرت بايثون 3 ببطء نتيجةً لعدم توافقيتها مع بايثون 2، مما يعني أنَّ على المستخدمين اختيار ما هو الإصدار الذي عليهم استخدامه. بالإضافة إلى ذلك، كانت الكثير من المكتبات البرمجية متاحةً فقط لبايثون 2، لكن بعد تقرير فريق تطوير بايثون 3 أنَّه يجب أن التخلي عن دعم بايثون 2، فبدأت عملية تحويل المكتبات إلى بايثون 3. يمكننا معرفة زيادة الاعتماد على بايثون 3 من خلال عدد الحزم البرمجية التي تدعم بايثون 3، والتي هي (في وقت كتابة هذا الكتاب) 339 من أصل 360 من أشهر الحزم.

3. بايثون 2.7

بعد إصدار بايثون 3.0 في 2008، أُصدِرَت نسخة بايثون 2.7 في تموز 2010 وهي آخر إصدار من سلسلة 2.x، الغرض من إصدار بايثون 2.7 هو جعل الطريق ممهَّدًا أمام مستخدمي بايثون 2.x لتحويل برامجهم إلى بايثون 3 بتوفير بعض التوافقية بينهما. وهذه التوافقية تضمنت دعم بعض الوحدات المُحسّنة في 2.7 مثل `unittest` لأتمتة الاختبارات، و `argparse` لتفسير خيارات سطر الأوامر، وبعض الفئات في `collections`. ولخصوصية بايثون 2.7 ولكونها جسرًا واصلًا بين الإصدارات القديمة من بايثون 2 وبين بايثون 3.0، فأصبحت خيارًا شائعًا بين المبرمجين بسبب توافقيتها مع الكثير من المكتبات.

عندما نتحدث اليوم عن بايثون 2، فنحن نشير عادةً إلى إصدار بايثون 2.7 لأنَّه أكثر إصدار مستخدم؛ لكنه يُعدُّ أنَّه إصدارٌ قديم، وسيتوقف تطويره (التطوير الحالي هو إصلاح العلل فقط) تمامًا في 2020.

4. الاختلافات الأساسية بين الإصدارات

بغض النظر أنَّ بايثون 2.7 وبايثون 3 تتشاركان في الكثير من الأشياء، لكن لا يجدر بك أن تظن أنَّهما متماثلتان ويمكن تبديل الشيفرات بينهما. ورغم أنَّك تستطيع كتابة شيفرات جيدة وبرامج مفيدة في أيِّ إصدار منهما، لكن من المهم أن تفهم أنَّ هنالك بعض الاختلافات في بنية الشيفرات وفي طريقة تفسيرها. سأعرض هنا بعض الأمثلة، لكن عليك أن تعلم أنَّك ستواجه المزيد من الاختلافات أثناء مسيرة تعلمك لبايثون.

أ. print

في بايثون 2، تُعامَل `print` معاملة التعليمات البرمجية (statement) بدلًا من كونها دالة، وهذا كان يثير ارتباكًا، إذ تتطلب الكثير من الأمور داخل بايثون تمرير وسائط (arguments) بين قوسين، إذا فتحتَ مُفسِّر بايثون 2 لطباعة "Sammy the Shark is my favorite sea creature"، فستكتب التعليمة `print` الآتية:

```
print "Sammy the Shark is my favorite sea creature"
```

أمَّا في بايثون 3، فستُعامَل `print()` معاملة الدوال، لذا لطباعة السلسلة النصية السابقة، فيمكننا استخدام شكل استدعاء الدوال التقليدي كما يلي:

```
print("Sammy the Shark is my favorite sea creature")
```

هذا التعديل جعل من البنية اللغويَّة في بايثون موحدةً وسهَّلًا من التبديل بين مختلف دوال الطباعة فيها. يجدر بالذكر أنَّ الدالة `print()` متوافقة مع بايثون 2.7، لذا ستعمل شيفرات بايثون التي تستعمل `print()` عملًا صحيحًا في أيِّ الإصدارَين.

ب. قسمة الأعداد الصحيحة

في بـايثون 2، أيُّ عـددٍ تكتبـه دون فواصـل عشـرية سـيُعامَل على أنَّه من النـوع `integer`، تأتي الإشكاليَّة عندما تحاول قسمة الأعداد الصحيحة على بعضها، فتتوقع في بعض الأحيان حصولك على عـددٍ عشـري (تسـمى أيضًا بالأعـداد ذات الفاصلة [`float`]) كما في العمليـة الرياضية التالية:

```
5 / 2 = 2.5
```

لكنَّ الأعداد الصحيحة في بايثون 2 لن تتحوَّل إلى أعداد عشريَّة عندما تتطلب العمليَّة التي تُجرَى عليها ذلك. عندما يكون العددان الموجودان على جانبَي معامل القسمة / عددين صحيحين، فستُجري بايثون 2 عمليَّة القسم وستُنتِج عددًا عشريًا إلا أنَّها ستُعيد العدد الصحيح الأصغر أو المساوي للناتج، وهذا يعني أنَّه لو كتبتَ `2 / 5` فستُعيد بايثون 2.7 العدد الصحيح الأصغر أو المساوي للعدد 2.5، وهو في هذه الحالة 2:

```
a = 5 / 2
print a

2
```

لإعادة عدد عشري، فيجب إضافة فواصل عشريَّة إلى الأرقام التي ستُجري عليها عملية القسمة كما في `2.0 / 5.0` لكي تحصل على النتيجة المنطقية `2.5`. أما في بايثون 3، فقسمة الأعداد الصحيحة أصبحت كما نتوقع:

```
a = 5 / 2
print a  # 2.5
```

يمكنك استخدام `2.0 / 5.0` لإعادة `2.5`، لكن إن أردتَ تقريب ناتج القسمة فاستخدم المعامل `//` الموجود في بايثون 3، كالآتي:

```
a = 5 // 2
print a

2
```

هذا التعديل في بايثون 3 جعل من قسمة الأعداد الصحيحة أمرًا سهلًا، لكن هذه الميزة غير متوافقة مع بايثون 2.7.

ج. دعم محارف يونيكود

عندما تتعامل لغات البرمجة مع السلاسل النصية (strings، والتي هي سلسلةٌ من المحارف)، فهي تفعل ذلك بطرائق مختلفة لكي تتمكن الحواسيب من تحويل الأعداد إلى أحرف ورموز.

تستعمل بايثون 2 محارف ASCII افتراضيًا، لذا عندما تكتب `"Hello, Sammy!"` فستتعامل بايثون 2 مع السلسلة النصية على أنَّها مجموعةٌ من محارف ASCII، والتي هي محدودةٌ لحوالي مئتَي محرف، أي أنَّ محارف ASCII هي طريقة غير عملية لترميز المحارف خصوصًا المحارف غير اللاتينية (مثل العربية مثلا). إن أردت استخدام ترميز محارف يونيكود (Unicode) الذي يدعم أكثر من 128000 محرف تابع للكثير من اللغات والرموز، فعليك أن تكتب `u"Hello, Sammy!"` إذ تُشير السابقة `u` إلى Unicode.

تستعمل بايثون 3 محارف يونيكود (Unicode) افتراضيًا، مما يوفِّر عليك بعض الوقت أثناء التطوير، ويمكنك كتابة وعرض عدد أكبر بكثير من المحارف في برنامجك بسهولة. يدعم يونيكود الكثير من المحارف بما في ذلك الوجوه التعبيرية (emojis)، واستعمالها ترميز محارف

افتراضي يعني أنَّ الأجهزة المحمولة ستكون مدعومةً في مشاريعك تلقائيًا. إذا كنت تحب أنَّ تكون شيفرات بايثون 3 التي تكتبها متوافقةً مع بايثون 2 فضع الحرف u قبل السلاسل النصية.

د. استمرار التطوير

الفارق الرئيسي بين بايثون 3 وبايثون 2 ليس في البنية اللغوية وإنما في أنَّ إصدار بايثون 2.7 توقف دعمه في 2020، وسيستمر تطوير بايثون 3 بميزاتٍ جديدة وإصلاحٍ لمزيدٍ من العلل. التطويرات الأخيرة في اللغة تتضمن تخصيصًا أبسط لإنشاء الأصناف، وطريقةً أوضح للتعامل مع المصفوفات... الاستمرار بتطوير بايثون 3 يعني أنَّ المطورين يمكن أن يعتمدوا على اللغة، وسيطمئنون أنَّ المشاكل التي قد تحدث فيها ستُحَل في فترةٍ قريبة، ويمكن أن تصبح البرامج أكثر كفاءة بإضافة المزيد من الميزات للغة.

5. نقاط أخرى يجب أخذها بالحسبان

عليك أن تضع النقاط الآتية بالحسبان عندما تبدأ مشوار البرمجة بلغة بايثون، أو عندما تبدأ بتعلم لغة بايثون بعد تعلمك لغيرها. إذا كنتَ تأمل بتعلم اللغة دون أن تفكِّر بمشروعٍ معيَّن، فأنصحك بالتفكير بمستقبل بايثون، فسيستمر تطوير ودعم بايثون 3 بينما سيوقف دعم بايثون 2.7 عمّا قريب (إن لم يكن قد توقف :-D). أمَّا إذا كنتَ تُخطِّط للانضمام لفريق تطوير أحد المشاريع، فعليك أن تنظر ما هو إصدار بايثون المستخدم فيه، وكيف يؤدي اختلاف الإصدار إلى اختلاف طريقة تعاملك مع الشيفرات، وإذا ما كانت المكتبات البرمجية المستعملة في المشروع مدعومةً في مختلف الإصدارات، وما هي تفاصيل المشروع نفسه.

إذا كنت تُفكِّر ببدء أحد المشاريع، فيجدر بك أن تنظر ما هي المكتبات المتوفرة وما هي إصدارات بايثون المدعومة. وكما قلنا سابقًا، الإصدارات الأوليَّة من بايثون 3 لها توافقيَّة أقل مع

المكتبات المبنية لبايثون 2، لكن الكثير منها قد جرى تحويله إلى بايثون 3، وسيستمر ذلك في السنوات الأربع المقبلة.

6. ترحيل شيفرة بايثون 2 إلى بايثون 3

بعد أن تعرفنا على إصدارات بايثون والفروق الجوهرية فيما بينها، سنتعلم الآن أفضل آليات وممارسات ترحيل الشيفرات من بايثون 2 إلى بايثون 3، وما إن كان عليك جعل الشيفرة متوافقة مع كلا الإصدارين.

وجدنا أنَّ الإصدار 2 من بايثون صدر عام 2000 ليُدشِّن حقبةً جديدةً من التطوير تقوم على الشفافية والشمولية، إذ شمل هذا الإصدار العديد من الميزات البرمجية، واستمر في إضافة المزيد طوال مدة تطويره.

يُعدُّ إصدار بايثون 3 مستقبل بايثون، وهو إصدار اللغة قيد التطوير حاليًا، فجاء في أواخر عام 2008، ليعالج العديد من عيوب التصميم الداخلية ويُعدِّلها. بيْد أنَّ اعتماد بايثون 3 كان بطيئًا بسبب عدم توافقه مع بايثون 2.

في خضم ذلك، جاء الإصدار بايثون 2.7 في عام 2010 ليكون آخر إصدارات بايثون 2.x وليُسهِّل على مستخدمي بايثون 2.x الانتقال إلى بايثون 3 من خلال توفير قدر من التوافق بين الاثنتين، فهذا هو الهدف الأساسي من إطلاقه.

ا. ابدأ ببايثون 2.7

للانتقال إلى بايثون 3، أو لدعم بايثون 2 وبايثون 3 معًا، يجب عليك التأكد من أنَّ شيفرة بايثون 2 متوافقة تمامًا مع بايثون 2.7.

يعمل العديد من المطورين حصريًا بشيفرات بايثون 2.7، أمَّا المبرمجون الذي يعملون بشيفرات أقدم، فعليهم أن يتأكدوا من أنَّ الشيفرة تعمل جيدًا مع بايثون 2.7، وتتوافق معه.

التأكُّد من توافق الشيفرة مع بايثون 2.7 أمرٌ بالغ الأهميَّة لأنَّه الإصدار الوحيد من بايثون 2 الذي ما يزال قيد الصيانة، وتُصحَّحُ ثغراته (قد توقف دعمه في وقت قراءتك لهذا الكتاب). فإذا كنت تعمل بإصدار سابق من بايثون 2، فستجد نفسك تتعامل مع مشكلات في شيفرة لم تعد مدعومة، ولم تعد ثغراتها تُصحَّح. هناك أيضًا بعض الأدوات التي تسِّهل ترحيل الشيفرة، مثل الحزمة Pylint التي تبحث عن الأخطاء البرمجية، لكن لا تدعمها إصدارات بايثون السابقة للإصدار 2.7.

من المهم أن تضع في حسبانك أنَّه رغم أنَّ بايثون 2.7 ما زالت قيد الدعم والصيانة في الوقت الحالي، إلا أنَّها ستموت في النهاية. ستجد في PEP 373 تفاصيل الجدول الزمني لإصدار بايثون 2.7، وفي وقت ترجمة هذا الكتاب، فإنَّ أجل بايثون 2.7 حُدِّد في عام 2020 (يحتمل أن تكون قد ماتت وأنت تقرأ هذه السطور :-( ).

ب. الاختبار

اختبار الشيفرة جزءٌ أساسيٌّ من عملية ترحيل شيفرة بايثون 2 إلى بايثون 3. فإذا كنت تعمل على أكثر من إصدار واحد من بايثون، فعليك أيضًا التحقُّق من أنَّ أدوات الاختبار التي تستخدمها تغطي جميع الإصدارات للتأكُّد من أنَّها تعمل كما هو متوقع.

يمكنك إضافة حالات بايثون التفاعلية (interactive Python cases) إلى السلاسل النصية التوثيقيَّة (docstrings) الخاصة بكافة الدوال والتوابع والأصناف والوحدات ثم استخدام الوحدة `doctest` المُضمَّنة للتحقق من عملها كما هو موضح إذ يعدُّ ذلك جزءًا من عملية الاختبار.

إلى جانب doctest، يمكنك استخدام الحزمة package.py لتتبع وحدة الاختبار. ستراقب هذه الأداة برنامجك وتحدد الأجزاء التي تُنفِّذها من الشيفرة، والأجزاء التي يمكن تنفيذها ولكن لم تُنفَّذ. يمكن أن تطبع Cover.py تقريرًا في سطر الأوامر أو تنتج مستند HTML. تُستخدم عادةً لقياس فعالية الاختبارات، إذ توضح الأجزاء من الشيفرة التي اختُبِرت، والأجزاء التي لم تُختبَر.

تَذكَّر أنه ليس عليك اختبار كل شيء، لكن تأكَّد من تغطية أيّ شيفرة غامضة أو غير عادية. للحصول على أفضل النتائج، يُنصح أن تشمل التغطية 80٪ من الشيفرة.

7. تعرف على الاختلافات بين بايثون 2 و بايثون 3

سيمكِّنك التعرُّف على الاختلافات بين بايثون 2 و بايثون 3 من استخدام الميزات الجديدة المتاحة، أو التي ستكون متاحة في بايثون 3.

تطرقنا مسبقًا إلى بعض الاختلافات الرئيسية بين الإصدارين، ويمكنك أيضًا مراجعة توثيق بايثون الرسمي لمزيد من التفاصيل.

عند البدء في ترحيل الشيفرة، فهناك بعض التغييرات في الصياغة عليك تعديلها فوريًا.

- Print: حلت الدالة print() في بايثون 3 مكان التعليمة print في بايثون 2.

```
# 2 بايثون
print "Hello, World!"

# 3 بايثون
print("Hello, World!")
```

- exec: تغيَّرت التعليمة exec في بايثون 2 وأصبحت دالةً تسمح بتمرير متغيرات محلية (locals) وعامة (globals) صريحة في بايثون 3.

```
# 2 بايثون
exec code                        # 1

exec code in globals             # 2

exec code in (globals, locals)   # 3

# 3 بايثون
exec(code)                       # 1

exec(code, globals)              # 2

exec(code, globals, locals)      # 3
```

- / و //: تُجرِي بايثون 2 القسمة التقريبية (floor division) بالعامل /، بينما تخصص بايثون 3 العامل // لإجراء القسمة التقريبية.

```
# 2 بايثون
5 / 2 = 2

# 3 بايثون
5 / 2 = 2.5
5 // 2 = 2
```

لاستخدام هذين العاملين في بايثون 2، استورد division من الوحدة `__future__`:

```
from __future__ import division
```

- `raise`: في بـايثون 3، يتطلب إطلاق الاسـتثناءات ذات الوسـائط اسـتخدام الأقـواس، كما لا يمكن استخدام السلاسل النصية كاستثناءات.

```
# با يثون 2
raise Exception, args                              # 1

raise Exception, args, traceback                   # 2

raise "Error"                                      # 3

# با يثون 3
raise Exception                                    # 1
raise Exception(args)

raise Exception(args).with_traceback(traceback)    # 2

raise Exception("Error")                           # 3
```

- `except`: في بايثون 2، كان من الصعب إدراج الاستثناءات المُتعدِّدة لكن ذلك تغيَّر في بايثون 3. لاحظ أنَّ `as` تُستخدَم صراحةً مع `except` في بايثون 3:

```
# با يثون 2
except Exception, variable:

# با يثون 3
except AnException as variable:
except (OneException, TwoException) as variable:
```

- `def`: في بـايثون 2، يمكن للـدوال أن تقبـل سلاسـل مثـل الصـفوف أو القـوائم. أمَّا في بايثون 3، فقد أزيل هذا الأمر.

```
# بايثون 2
def function(arg1, (x, y)):

# بايثون 3
def function(arg1, x_y): x, y = x_y
```

- `expr`: لم تعد صياغة علامة الاقتباس المائلة `` ` ` `` في بايثون 2 صالحة، واستخدم بدلًا عنها `repr()` أو `str.format()` في بايثون 3.

```
# بايثون 2
x = `355/113`

# بايثون 3
x = repr(355/113):
```

- تنسيق السلاسل النصية (String Formatting): لقد تغيرت صياغة تنسيق السلاسل النصية من بايثون 2 إلى بايثون 3.

```
# بايثون 2
"%d %s" % (i, s)                           # 1
"%d/%d=%f" % (355, 113, 355/113)           # 2

# بايثون 3
"{} {}".format(i, s)                       # 1
"{:d}/{:d}={:f}".format(355, 113, 355/113)  # 2
```

تذكر كيفية استخدام تنسيقات السلاسل النصية من فصل كيفية استخدام آلية تنسيق السلاسل النصية في بايثون 3.

- `class`: ليست هناك حاجة لتمرير `object` في بايثون 3.

```
# بايثون 2
class MyClass(object):
   pass

# بايثون 3
class MyClass:
   pass
```

في بايثون 3، تُضبَط الأصناف العليا (metaclasses) بالكلمة مفتاحية `metaclass`.

```
# بايثون 2
class MyClass:
   __metaclass__ = MyMeta
class MyClass(MyBase):
   __metaclass__ = MyMeta

# بايثون 3
class MyClass(metaclass=type):
    pass
class MyClass(MyBase, metaclass=MyMeta):
    pass
```

8. تحديث الشيفرة

هناك أدَاتان رئيّسيتان لتَحديث الشيفرة تلقائيًا إلى بايثون 3 مع الحفاظ على توافقيَّتها مع بايثون 2 وهما: future و modernize. تختلف آليَتا عمل هاتين الأداتين، إذ تحاول `future` نقل أفضل ممارسات بايثون 3 إلى بايثون 2، في حين أنَّ `modernize` تسعى إلى إنشاء شيفرات موحَّدة لبايثون تتوافق مع 2 و 3 وتستخدم الوحدة `six` لتحسين التوافقيَّة. يمكن أن تساعدك هاتان الأداتان في إعادة كتابة الشيفرة وتحديد ورصد المشاكل المحتملة وتصحيحها. يمكنك تشغيل الأداة عبر مجموعة unittest لفحص الشيفرة والتحقُّق منها بصريًا، والتأكد من أنَّ

المراجعــات التلقائيــة الــتي أُجــريَت دقيقــة. وبمجــرد انتهــاء الاختبــارات، يمكنــك تحويل الشيفرة.

بعد هذا، ستحتاج على الأرجح إلى إجراء مراجعة يدويَّة، وخاصة استهداف الاختلافات بين بايثون 2 و 3 المذكورة في القسم أعلاه. إن أردت استخدام future، فعليك إضافة عبارة الاستيراد التالية في جميع وحدات بايثون 2.7:

```
from __future__ import print_function, division,
absolute_imports, unicode_literals
```

رغم أنَّ هذا لن يعفيك من إعادة كتابة الشيفرة، إلا أنَّه سيضمن لك أن تتماشى شيفرة بايثون 2 مع صياغة بايثون 3.

أخيرًا، يمكنك استخدام الحزمة pylint لتحديد ورصد أي مشكلات محتملة أخرى في الشيفرة. تحتوي هذه الحزمة على مئات القواعد التي تغطي مجموعة واسعة من المشكلات التي قد تطرأ، بما فيها قواعد الدليل PEP 8، بالإضافة إلى أخطاء الاستخدام.

قد تجد أنَّ بعض أجزاء شيفرتك تربك pylint وأدوات الترحيل التلقائي الأخرى. حاول تبسيطها، أو استخدم unittest.

9. التكامل المستمر (Continuous Integration)

إذا أردت أن تجعل شيفرتك متوافقة مع عدة إصدارات من بايثون، فستحتاج إلى تشغيل الإطار unittest باستمرار وفق مبدأ التكامل المستمر (وليس يدويًا)، أي أن تفعل ذلك أكبر عدد ممكن من المرات أثناء عملية التطوير. إذا كنت تستخدم الحزمة six لصيانة التوافقيَّة بين بايثون 2 و 3، فستحتاج إلى استخدام عدَّة بيئات عمل لأجل الاختبار.

إحدى حزم إدارة البيئة التي قد تكون مفيدة لك هي الحزمة tox، إذ ستفحص تثبيتات الحزمة مع مختلف إصدارات بايثون، وإجراء الاختبارات في كل بيئة من بيئات عملك، كما يمكن أن تكون بمثابة واجهة عمل للتكامل المستمر.

10. خلاصة الفصل

لغة بايثون كبيرة جدًا وموثقة توثيقًا ممتازًا وسهلة التعلم، ومهما كان اختيارك (بايثون 2 أو بايثون 3) فستتمكن من العمل على المشاريع الموجودة حاليًا. صحيحٌ أنَّ هنالك بعض الاختلافات المحورية، لكن ليس من الصعب الانتقال من بايثون 3 إلى بايثون 2، وستجد عادةً أنَّ بايثون 2.7 قادرة على تشغيل شيفرات بايثون 3، خصوصًا في بدايات تعلمك للغة. من المهم أن تبقي ببالك أنَّ تركيز المطورين والمجتمع أصبح منصبًّا نحو بايثون 3، وستصبح هذه اللغة رائدةً في المستقبل وستلبي الاحتياجات البرمجية المطلوبة، وأنَّ دعم بايثون 2.7 سيقل مع مرور الزمن إلى أن يزول في 2020 (قد زال لحظة ترجمة الكتاب ومراجعته).